A TRIVIUM-INSPIRED PSEUDORANDOM NUMBER GENERATOR WITH A STATISTICAL COMPARISON TO THE RANDOMNESS OF SECURERANDOM AND TRIVIUM

Latoya Niesha Jackson

2017

Columbus State University

D. Abbott Turner College of Business

TSYS School of Computer Science

The Graduate Program in Applied Computer Science

A Trivium-Inspired Pseudorandom Number Generator

with a Statistical Comparison to the Randomness of SecureRandom and Trivium

A Thesis in

Applied Computer Science

by

Latoya Niesha Jackson

Submitted In Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2017

Abstract

A pseudorandom number generator (PRNG) is an algorithm that produces a sequence of numbers which emulates the characteristics of a random sequence. In comparison to its genuine counterpart, PRNGs are considered more suitable for computing devices in that they do not consume a lot of resources (in terms of memory) and their portability; they can also be used on a wide range of devices. Cryptographically Secure PRNGs (CSPRNGs) are the only type of PRNGs suitable for cryptographic applications. They are specially designed to withstand security attacks. In this thesis, we provide descriptions of two CSPRNGs: Trivium, a hardware-based stream cipher designed for constrained computing environments, and OpenJDK SecureRandom, a traditional CSPRNG recommended for Java programs that include a cryptographic algorithm. Our contributions are Quadrivium, a PRNG inspired by Trivium and analysis results comparing statistical properties of SecureRandom, Trivium and Quadrivium.

Index Words: Cryptographically secure pseudorandom number generators, Lightweight cryptography, SecureRandom, Trivium

Table of Contents

List of Tables

List of Figures

The generation of random numbers is too important to be left to chance.

– Robert R. Coveyou

ACKNOWLEDGEMENTS

First, I would like to thank my thesis adviser, Dr. Yesem Peker. You have been extremely accommodating throughout this entire process, making yourself available for guidance and support at all times. I greatly appreciate how you welcome my creativity and visions with enthusiasm and allowed this thesis to be my own work. This has truly been an enriching experience for me and I owe it all to you. I would also like to extend my gratitude to the thesis committee: Dr. Radhouane Chouchane, Dr. Rodrigo Obando and Dr. Lydia Ray. Your dedication to learning has made this research possible. Furthermore, without your expertise or assistance, this paper could not have been completed successfully. Again, I am grateful for your efforts. To my dear family and friends who have always believed in me and been my personal cheerleaders, thank you. I am almost there. Lastly, I want to give a special thanks to my sister, Shona. This paper, this journey is as much yours as it is mine. Thank you for everything, Sis.

Chapter 1

An Introduction to Pseudorandomness

Randomness is defined as "the quality or state of lacking a pattern or principle of organization; unpredictability."[1] Essentially, it is the inability to predict what will happen next even though there is knowledge of what has happened before. Despite the chaotic disposition of randomness, it has made itself a necessity in many orderly processes since it fosters diversity, fairness, creation and security.

Randomness has a presence in areas such as Monte Carlo simulation, statistical sampling, gaming, internet gambling and cryptography. Monte Carlo simulation methods employ randomness to solve optimization, numerical integration and probability distribution problems. In statistical sampling, randomness is used to help select arbitrary samples for analysis. Computer-controlled characters and procedural generation in electronic gaming also uses randomness as a source of variability. Internet gambling requires a source of unpredictability to ensure game integrity and combat cheating. In cryptography, randomness is implemented by generating secret keys for well-known ciphers such as AES, RSA and Blowfish; it is used to encrypt messages for One Time Pads or to conceal information in protocols by converting the data to seemingly random sequences.

1. 1 Theories of Randomness

Across various disciplines, randomness is defined differently but there are three accepted theories. The first theory of randomness originated in Information theory. The second theory of randomness is rooted in the concept of universal language which is a part of computability theory. The third theory of randomness, which is the foundation of this thesis, is based on principles in complexity theory.

The first theory of randomness, rooted in information theory, was presented by mathematician Claude E. Shannon in 1948. It came as a result to his model which mimics the flow of information. The structure consists of an information source, transmitter, channel, receiver and destination. The information source is a randomized object responsible for producing a message or a string of messages. All messages produced are from a finite discrete set of messages. The elements of this set are known.

Shannon was concerned with saving time transmitting a string of messages through a channel. As a result, the transmitter dispatches a string with missing values to the receiver. Shannon believed the receiver can reconstruct the original string from the transmitted string, statistically. This is done by using points (messages) within the string to determine missing values. However, this led to another concern of Shannon: quantifying the amount of uncertainty present in the receiver reconstructing the initial string from a randomized source. In this sense, Shannon equated randomness to uncertainty. Additionally, randomness is viewed as a probabilistic entity since possible selected messages are restricted to an equally probable set of elements.

---

[1] Oxford Dictionary

Shannon's measure of uncertainty, entropy, is on a continuous set from zero to one. A zero entropy value signifies that the probability of a selected message is certain. An entropy value of one denotes the most uncertain (or perfectly random) case.

The second theory of pseudorandomness was independently developed by Ray Solomonoff (1960/1964), Andrey Kolmogorov (1965) and Gregory Chaitin (1969). It is commonly referred to as Kolmogorov Complexity. Like Shannon's information theory, Kolmogorov Complexity proposes a practical measure of information from a random source. Contrastingly, the theory pursues an algorithmic approach and describes, rather than quantifies, information.

Kolmogorov Complexity considers a binary string (from an information source) and defines it by the most concise method required to reconstruct it. A more in depth perspective looks at the method as some function; it takes an input of length x to produce y, the initial string. How far the initial string can be compressed gauges its randomness. An initial string that can only be described as itself lacks structure. Hence, Kolmogorov Complexity views randomness as a lack of structure.

Unlike the first two theories, the third theory of randomness is motivated by a completely different scenario. The first and second theories are concerned with expressing information that shows regularity. Additionally, perfect randomness was considered extreme cases for both notions. The third theory of randomness, pseudorandomness, seeks to express information that achieves perfect randomness. This theory is grounded in complexity theory and views randomness in relation to an observer's analytical abilities. Thus, if an object appears random to an observer then it is random. Let us demonstrate this viewpoint with a game of heads or tails with Alice and Bob. A fair coin is used and the game proceeds under different conditions.

In the first scenario, Bob declared his choice then Alice flips the coin. Here, Bob has a probability of one half to correctly guess how the coin lands.

In the second scenario, Alice flips the coin then in mid-air Bob declares his choice. Like the first scenario, Bob has a fifty-fifty chance of winning.

In the third scenario, Alice flips the coin. But this time, Bob has a machine that is able to determine how the coin will land based on its motion in air and other external effects. The problem with this machine is it cannot compute all this information in time for Bob to declare his choice. Bob still has a probability of one half to make a correct guess.

In the fourth scenario, Bob has an efficient machine that is able to make all the necessary computations required to help Bob declare his choice in time. Alice flips the coin and in midair Bob makes a call based on the prediction he received. Bob's chances of winning in this situation have increased making this a biased game. [Gol10]

1.2 Implementing Randomness

For the applications stated in the beginning of the chapter, randomness is implemented as nonlinear sequences comprised of binary bits. Truly random bits are very expensive and arduous to attain. They require a physical source, which may possibly be biased and/or not available for large scale distribution. Some examples of physical sources are radioactive decay, thermal noise

or atmospheric noise. PRNGs offer a more practical method to obtain "random" sequences. According to the principles of the third theory of randomness, pseudorandom values are as efficient as its real counterpart providing good substitutes for truly random quantities whenever needed. Please take note that not all pseudorandom sequences are created equally. Therefore, some values should not be used as a surrogate for truly random sequences. This holds true in the area of cryptography. We will review various PRNG properties suitable for cryptographic applications. Furthermore, we will use these findings to construct our own generator, Quadrivium.

1.3 Outline

In the subsequent chapter, we will discuss pseudorandom number generators, its general structure, fundamental aspects, and other relative information pertaining to PRNGs. We will look at a special type of generators, cryptographically secure PRNGs in chapter three. We will also discuss the factors that discriminate between a non-cryptographic generator and a cryptographic generator. Chapters 4 and 5 give a description of OpenJDK SecureRandom, the official CSPRNG implementation of Java SecureRandom Class and Trivium, a stream cipher whose keystream may be used as a CSPRNG, respectively. Our contributions will be presented in the sixth and seventh chapters. First, a description of our Trivium-based PRNG, Quadrivium, will be provided. Then, we will show analyses of OpenJDK SecureRandom, Trivium and Quadrivium.

Chapter 2

Pseudorandom Number Generators

2.1 Background (Definitions)

Before we proceed, there are some standard definitions we will first discuss regarding pseudorandom number generators.

Definition (discrete probability distribution) Given a discrete set of random variables, a discrete probability distribution is a function denoting the likelihood of random variable n occurring in an event such that the sum of all probabilities of all n in the set equals one.

Definition (discrete uniform distribution) A discrete uniform distribution is a probability distribution such that on a finite set of n variables, the occurrence of any variable is $\frac{1}{n}$.

Definition (probability ensemble) Let X_n be a probability distribution. An ensemble X is a sequence of probability distributions such that

$$X := \{X_n\}_n, \text{ where } n \in \mathbb{N}.$$

Definition (uniform ensemble) Let U_n be a discrete uniform distribution over the set $\{0, 1\}^n$. A uniform ensemble U is defined as

$$U := \{U_n\}_n, \text{ where } n \in \mathbb{N}.$$

Definition (pseudo-random number generator) The algorithm G is a pseudorandom number generator if

$$G := \{0,1\}^m \rightarrow \{0,1\}^n,$$

where $m << n$ and $G(U_m)$ is (t, ε)-computationally indistinguishable from U, for some large t and negligible ε. We will see in later in this chapter what it is meant to be "computationally indistinguishable."

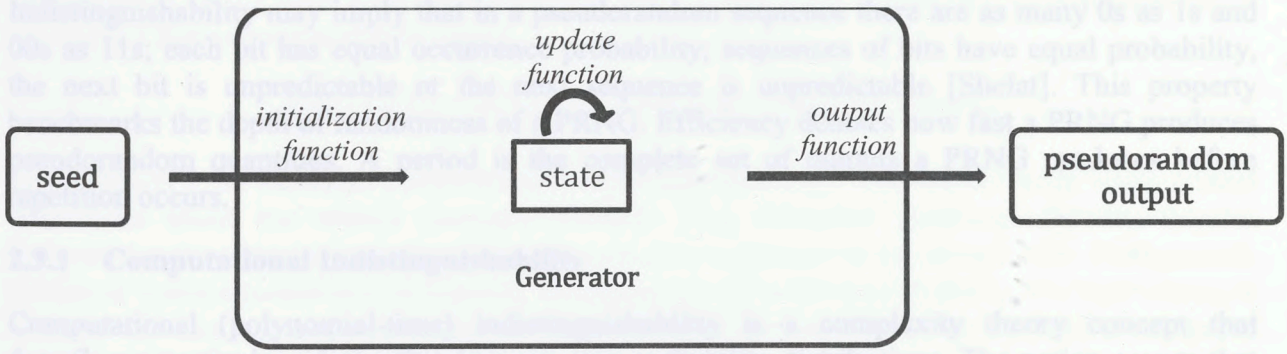

Figure 1. Diagram of a pseudorandom number generator.

2.2 Fundamental Aspects of PRNGs

There are three fundamental aspects of pseudorandom number generators: the generating process, relative distinguishers and computational complexity. The first aspect pertains to the basic components that make a PRNG function. Relative distinguishers pertain to objects used to differentiate pseudorandom quantities from truly random quantities. Lastly, computational complexity is concerned with the hardness of the generating algorithm. [Gol10]

The generating procedure of a PRNG requires an algorithm, a seed and an output. The seed is typically a small quantity of truly random bits. The algorithm takes the seed and outputs a longer sequence of pseudorandom bits. (Refer to Figure 1.)

Distinguishers are any algorithm or code capable of discriminating between pseudorandom quantities and truly random quantities. Consequently, they are viewed as challengers to PRNGs. Due to their range and depth of abilities (prediction, time efficiency, etc); distinguishers are partitioned into varying classes. The quality of a generator is gauged by the types of distinguishers it can withstand. (Quality encompasses predictability, randomness and periodicity of the pseudorandom outputs.) Additionally, distinguishers run in polynomial time and may be more algorithmically complex than the generators they challenge. This aspect is associated with the desirable property of indistinguishability.

Computational complexity relates to resources used to solve or understand the workings of the generating algorithm. It is similar to algorithmic analysis but focuses more on resource availability. Given a restricted number of resources, how difficult is it to analyze an algorithm. On the other hand, algorithmic analysis considers the number of resources required to analyze an algorithm. The type of resources here are time and memory storage, or in other words, the time complexity and space complexity. It is greatly desired that the generator operate in polynomial time.

2.3 Desirable Properties

Desirable properties of a PRNG include indistinguishability, efficiency and a long period. Indistinguishability may imply that in a pseudorandom sequence there are as many 0s as 1s and 00s as 11s; each bit has equal occurrence probability; sequences of bits have equal probability, the next bit is unpredictable or the next sequence is unpredictable [Shelat]. This property benchmarks the depth of randomness of a PRNG. Efficiency denotes how fast a PRNG produces pseudorandom quantities. A period is the complete set of outputs a PRNG produces before repetition occurs.

2.3.1 Computational Indistinguishability

Computational (polynomial-time) indistinguishability is a complexity theory concept that describes a particular relationship between two probability distributions. The notion asserts that two distinct probability distributions are computationally indistinguishable if they cannot be told a part by some discriminating method in polynomial time. [Gol90]

Definition (computational indistinguishability) Let $X \coloneqq \{X_n\}_{n\epsilon\mathbb{N}}$ and $Y \coloneqq \{Y_n\}_{n\epsilon\mathbb{N}}$ be probability ensembles over $\{0, 1\}^m$ and f some function. X and Y are (t, ε)-computationally indistinguishable if for every f that is computable in time t,

$$|Pr[f(X) = 1] - Pr[f(Y) = 1]| \leq \varepsilon, \text{ for some negligible value } \varepsilon .$$

Definition (pseudorandom ensemble) Let $X \coloneqq \{X_n\}_{n\epsilon\mathbb{N}}$ be a probability ensemble and $U \coloneqq \{U_n\}_{n\epsilon\mathbb{N}}$ be a uniform ensemble. X is a pseudorandom ensemble if it is computationally indistinguishable from U.

There are various methods involved to launch an attack on PRNGs. Computational indistinguishability is more concerned with the time factor involved in discriminating pseudorandom outputs from truly random outputs. The theory restricts these methods to the computational resources available for an attack. It assumes that an adversary only possess time efficient algorithms to launch an attack.

An algorithm is said to be time efficient if it operates in the same manner as a probabilistic polynomial-time Turing machine. Probabilistic polynomial-time Turing machines consist of the basic operations of a Turing machine but have an additional writing instruction. At each point, the machine can select a random variable to write based on the probability distribution under scrutiny. In actuality, this machine has an internal random selection process to make such decisions. This only explains the probabilistic nature of the computing system. As for the polynomial-time characteristic, the maximum number steps required to compute an input of length n can be expressed as a polynomial. If a PRNG can resist any time efficient algorithmic attack then, by computational indistinguishability, it is a sufficient random generator substitute. [Roe05]

2.3.2 Statistical Indistinguishability

Aside from computational indistinguishability, there exists a notion called statistical indistinguishability. The two differ in that one is grounded in polynomial time algorithms and the latter derives from data collection and analysis.

Definition (statistical indistinguishability) Let $X \coloneqq \{X_n\}_{n\epsilon\mathbb{N}}$ and $Y \coloneqq \{Y_n\}_{n\epsilon\mathbb{N}}$ be probability ensembles in $\{0, 1\}^m$ and S be some statistical test. X and Y are statistically indistinguishable if for every S that is computable in time t,

$$\max_{S\subseteq\{0,1\}^m} |S(X) - S(Y)| \leq \varepsilon, \text{ for some negligible value } \varepsilon .$$

In Goldreich's "A Note on Computational Indistinguishability," the author made some crucial deductions about the related concepts [Gol90]. One, statistical indistinguishability implies computational indistinguishability. Two, the converse statement is not always true. Furthermore, statistical indistinguishability is superior to computational indistinguishability. For that reason, it is fair to say that statistical indistinguishability is a more desirable property of a PRNG compared to computational indistinguishability.

Statistical indistinguishability suggests two probability distributions are similar if the statistical distance between the two is insignificant. Its formal definition is analogous to that of

computational indistinguishability but the functions represent some arbitrary statistical tests. Later in this literature, we will see the importance of this type of indistinguishability; many standard testing for randomness quality are influenced by statistical elements.

2.3.3 Efficiency and Periodicity

The desirable property of efficiency, in this case, refers to time complexity. For some tasks, like stream ciphers, simulations and protocol masking, a PRNG must produce rapidly. The reason being these tasks require random outputs of sizeable lengths. Efficiency and robustness usually do not pair well in a PRNG. Robust PRNGs demand highly complex algorithms which in turn lowers the PRNGs speed. [Roe05]

The importance of a long period is trivial. Remember von Neumann's middle square method. One of its limitations was that the outputs cycled over in a short space of time. A shorter period enables an attacker to fully grasp the workings of the internal state of a PRNG. This results in an effortless attempt to predict the past and future random outputs.

2.4 Some PRNG Algorithms

Pseudorandom generators can be traced back to 1951 with John von Neumann's middle square method. The algorithm produces a sequence by first taking an n-length sequence as the input value. That number is squared to obtain a $2n$-length number; leading zeroes are added to meet the required length. Then, the middle n-digits are extracted. To generate the next sequence, the extracted n-length sequence is used as the input value and undergoes the aforementioned steps. The middle square method is not an ideal generator since there are some sequences, if taken as the initial value, will produce an all zero sequence. Once this occurs, all future sequences will be all zeroes. Additionally, the sequences cycle over in a very short period of time. [Von 51]

2.4.1 Linear Congruence Method

This generator employs a piecewise linear equation to return random outputs. The construction of the method is simple and comprises of four values: an initial value, a multiplier, an increment and a modulus. The initial value, multiplier and increment are greater than or equal to zero; the modulus is strictly greater than the aforementioned numerals. The algorithm goes as follows:

$$X_{n+1} = (aX_n + b) mod\ c,$$

where X is a pseudorandom sequence, a is the multiplier, b is the increment and c is the modulus.

First, the product of the multiplier and the initial value are added to the increment. The result from the previous operation undergoes a modular arithmetic: *result* (mod *modulus*). The result from this operation yields a random output. To obtain the next random sequence, the current random sequence is used to initialize the generator. Additionally, when the increment equals zero, the algorithm is called a multiplicative congruential method; when the increment does not equal zero, the algorithm is called a mixed congruential method. [Xiannong]

2.4.2 Mersenne Twister

Generators that use linear recurrences are simple in nature. These generators are needed for applications where the predictability of outputs is not important. With this technique, each output is dependent on the previous output. These generators can be further categorized by their core algorithms. Feedback shift register is a subtype of linear recurrences. It is a shift register whose input bit is a transformation of its previous state. The main function is an exclusive or on individual bits. Feedback shift registers can be linear or twisted.

In 1997, Japanese mathematicians Makoto Matsumoto and Takuji Nishimura revealed the Mersenne Twister, a PRG that produces ideal random numbers at a fast generation rate. The generator is the most widely used PRG for general usage. The first part of the generator's name refers to its large period of $2^{199937} - 1$, which is one of the largest Mersenne Primes. The Twister component refers to the algorithm being a modified version of the Twisted Generalized Feedback Shift Register (TGFSR) [Jag 08]. TGFSR employs a linear recurrence equation in which the initial seed does not affect the quality of the random outputs, larger periods and greater speed than previous PRGs [MK 92]. The Mersenne Twister "takes an incomplete array to realize a Mersenne prime as its period and uses an inversive-decimation method for primitivity testing of a characteristic polynomial of a linear recurrence with a computational complexity of $O(p^2)$ where p is the degree of the polynomial" [Jag 08]. Moreover, random outputs from the algorithm appear to be uniformly distributed in 623 dimensions.

2.4.3 Xorshift

This pseudorandom generator was developed in 2003 by American mathematician George Marsaglia. (He is also the creator of Diehard Battery of Tests, a statistical testing suite for PRNGs.) Xorshift is a subset of linear feedback shift registers. Random outputs are created by a recursive mechanism which XORs a binary bit sequence with an n-bit shifted version of itself. The algorithm is notably fast and produces computationally indistinguishable random numbers. [Mar 03]

2.5 PRNG Failures

Some PRNG failures are attributed to the lack of entropy or how entropy is acquired within a system. Entropy is a collection of sources employed to update the internal state of a PRNG. Examples of sources include mouse movement, keystroke timing and noise from the system's soundcard. In what manner entropy impacts a generator varies between constructs. In the case of Linux /dev/random, a PRNG, the amount of entropy determines if the generator is fit for operation. An inadequate entropy level halts /dev/random from yielding outputs. Other failures may be based on short periodicity or linear complexity of the generating function.

Additionally, PRNG failures can be attributed to the limited knowledge on how different constructions staunchly affect the quality of computed outputs. No matter how strong a cryptosystem may be, its security is directly proportionate to the strength of its PRNG. Countless cryptosystems employ pseudorandom generators that are poorly designed. If that is not the case, the system implements the generator in such a way that makes the PRNG prone to failure.

Debian experienced a security breach with its OpenSSL distribution. The pseudorandom generator included in the implementation was incapable of acquiring high levels of entropy. This caused the PRNG to produce 32,767 distinct private keys. The small key space ensue highly predictable keys. Other Debian-based products, like Ubuntu, were affected by this PRNG failure. [Debian]

Another entropy-based case involves the internationally used MIFARE Classic chip. It has applications in contactless smart cards and proximity cards. Various academic attacks have been carried out on the chip to reveal its vulnerabilities. In a 2008 paper by de Koning Gans, Hoepman and Garcia, the authors disclosed weaknesses in the MIFARE Classic chip PRNG. The researchers were able to recover keystreams, read memory blocks and modify memory blocks from the chip. This was all due to the low entropy collecting PRNG implemented in MIFARE. [dHG08]

Security Socket Layer (SSL) uses a PRNG to generate a random key. The key is used in a cryptographic algorithm to encrypt information flowing between client and server. Netscape utilized its own implementation of SSL to protect transmission of sensitive data over its browser. However, two computer science students were able to decipher encrypted messages sent over Netscape Web. They uncovered flaws in the PRNG used in the Netscape SSL implementation. Encrypted messages became recoverable due to the three values that seeded the PRNG. The values —taken from the running system— are the process ID, the parent process ID and the time of day. These values are indeed unique but still predictable, hence, the key was retrievable as well as the messages. [GW96]

Shortly following a publication which analyzed the security of popular SecureRandom constructs, a Bitcoin incident occurred leaving its Android users vulnerable to theft[2] [MMS13]. The two events are related in that SecureRandom is a special PRNG for cryptographic applications and Android uses it for cryptographic Bitcoin procedures. However, the Android SecureRandom implementation had a bug that caused the generator to yield predictable sequences. The paper revealed how the generator produced colliding values—making the private key recoverable. The paper also discussed the PRNG's defects in entropy collection and the capability to overwrite the seed value.

2.6 Scholarly Attacks

From the PRNG failures aforementioned, we see that the information gained led to the recovery of the encryption key. This allowed the "attackers" to decrypt secret messages or have access to private data. Key recovery is not the only concern for PRNG failures. There also exists the possibility of influencing the output of a PRNG or acquiring knowledge of the internal state. If an attacker knows the internal state of a PRNG then the attacker is capable of predicting future outputs. In "Cryptanalytic Attacks on Pseudorandom Number Generators" [KSW+98], the work outlined some possible attacks on PRNGs to obtain the information aforementioned. We will limit our discussion to attacks stated in this paper. There are other documents available detailing additional attacks.

---

[2] https://bitcoin.org/en/alert/2013-08-11-android

The first attack described in [KSW+98] is the Direct Cryptanalytic Attack. The name stems from the adversary's ability to directly observe computed outputs, therefore, permitting the adversary to make distinctions between truly random values and generated random values. Outputs from the observed PRNG are collected. Consequently, the internal state is learned leading to possibility of predicting future outputs. Most PRNGs are vulnerable to these attacks since their outputs can be accessed during the transmission stage or other processes. Triple-DES PRNGs, for example, are not affected by the direct cryptanalytic attack because the generated key is protected under obscurity.

Another possible attack defined in the paper is the Input-Based Attack. This class of attacks is possible if an adversary can use the knowledge of PRNG inputs or manipulate the inputs to determine the next bit. Input-Based Attacks consists of three types: known input, chosen input and replayed input. Known Input-Based Attacks are based on inputs that, contrary to the belief of the PRNG author, are predictable to an attacker. For example, generators that use inputs sourced from disk latency are vulnerable. There is the possibility that the attacker can retrieve data about the timings. Adversaries that have knowledge of the inputs and outputs of the generator characterize the Chosen Input-Based Attack. This attack pertains to systems that utilized user-based inputs, known plaintext messages or network data as a source of entropy [KSW+96]. The attacker extracts information about the PRNG via the known data or adversary-selected data fed to the generator. The input-output pairings aid in learning about the internal state of the generator. The Replay Input-Based Attack is quite similar to the Chosen Input-Based Attack. This attack is only restricted to known inputs or inputs supplied by an external source. The adversary has no control on the entropy given to the PRNG.

State Compromise Extension Attacks occur when an adversary further exploits a compromised system. The adversary has already recovered the internal state of the PRNG and uses that to its advantage to recover outputs prior to the time since the system has been compromised. Furthermore, the attacker can predict or has adequate knowledge about future outputs. This is possible under the condition of a security breach, successful cryptanalysis or an unintentional data leak. State Compromise Extension attacks can be executed in various ways. One is termed the Permanent Compromise Attack. This occurs once the internal state has been learned resulting in an irreparable collapse of the PRNG. Both forward and backward secrecy no longer hold. Iterative Guessing Attack occurs when unknown but predictable PRNG inputs are collected at different points of time to learn the current state of the PRNG after a state compromise. Backtracking Attacks use the knowledge of the internal state at some arbitrary point of time to foretell future outputs. State Compromise Extension Attack is also possible under the condition in which a PRNG is initiated from low entropy. Another condition is if the internal state has been successfully covered from a Direct Cryptanalysis Attack or one of the Input-Based Attacks. The Meet-in-the-Middle attack coalesce principles from the Iterative Guessing Attack and the Backtracking Attack. With knowledge of a past compromised state and a future state, an attacker can specify the current state of a PRNG.

Chapter 3

Cryptographically Secure PRNGs

In cryptography, Kerckhoff's principle, which is also considered an axiom or law in the field, is the concept that a cryptosystem must maintain its security even if it falls into the hands of an adversary. In other words, an attacker may have an encrypted message in his possession and is knowledgeable of the encoding function used; however, the secrecy of the message will not be compromised. This is only possible if the key remains unknown. To ensure such secrecy, cryptographers use random numbers as keys. Random numbers are a reliable source of unpredictability; it is quite difficult for an attacker to recover a randomly selected key, especially if it is drawn from a large pool.

Weaknesses in the random generation process directly affect the strength of cryptographic algorithms and cause them to be susceptible to attacks. The only type of PRNGs that is suitable for cryptographic applications is CSPRNG, cryptographically secure pseudorandom number generator. It has the core components of a PRNG—a seed, generating function and pseudorandom output— as well as two additional components: an entropy source and an internal state. Entropy, as defined earlier, is the measure of disorder. Entropy sources add disorder to a generator and in turn aid in the quality of the random outputs as well as refresh the internal state. The internal state comprises of all stored values, such as parameters and variables, which a CSPRNG relies on to function.

Non-cryptographic pseudorandom number generators and cryptographically secure pseudorandom number generators essentially perform the same task: provide seemingly random data for applications in need. Outputs from the two can be used in domains such as gaming theory, approximation algorithms, counting problems in addition to property testing. However, only CSPRNGs can be used for cryptographic applications. Discriminating factors between the generators can be summed up in two notions: strength of distinguishers and unpredictability.

3.1 Distinguishers

The existence of distinguishers plays a significant role in categorizing PRNGs. As was discussed in the previous chapter, distinguishers are challengers of pseudorandom number generators. Some distinguishers are more complex than others and may have the ability to correctly predict the outputs of a pseudorandom number generator. The types of distinguishers a PRNG can withstand, determines if a PRNG is suitable for cryptographic applications.

The basic class of distinguishers describes probabilistic polynomial-time algorithms. First, polynomial-time algorithms are processes that can be completed in polynomial-time, or in other words, speedily. Probabilistic algorithms are procedures that have a built-in random process and use the result to make decisions. These types of distinguishers can decide if a sequence was taken from a set of purely random sequences or a set of pseudorandom sequences. These distinguishers are related to the notion of computational indistinguishability. Both cryptographic and non-cryptographic PRNGs have this property. [Gol08]

Another class of distinguishers is a set of statistical tests or statistically inclined algorithms. They examine pseudorandom sequences and declare if they exhibit expected behaviors of random sequences. Random sequences are said to have uniform distribution of all probable events—in most cases, binary bits. NIST STS, Diehard battery of tests and Dieharder, for short, are collections of statistical distinguishers. We will further discuss these collections of distinguishers in chapter 7.

Space bounded distinguishers are challengers that do not rely on computational assumptions. They are time bounded procedures with restricted space complexity. These distinguishers are limited in capability and function in an automata-like fashion.

3.2 Unpredictability

Non-cryptographic generators are created under a weaker assumption. They are only required to be statistically and computationally random. This means they only need to be indistinguishable from random by statistical properties and in polynomial time. Cryptographic generators must possess an additional property, unpredictability. [Sta11]

CSPRNG outputs serve as keys for cryptographic algorithms. The security of the keys relies on their random selection. CSPRNG outputs are deemed secure because their values are unpredictable. Also, the outputs do not reveal any information about the inner workings of the generator. In contrast, non-cryptographic PRNG sequences may exhibit patterns or behaviors disclosing the mechanism of its generating algorithm. Since CSPRNG outputs reveal no information, even in the event that the current sequence (or key) is known, it is impossible to uncover future or past sequences.

Unpredictability guarantees pseudorandom values produced by a generator lacks structure, cannot be controlled nor conform to some pattern. Unpredictability does not equate to true randomness but is another form of randomness that entails high entropy. It is considered more practical than perfect randomness—which is not accessible for all systems [Dodis]. Unpredictability implies that given a sequence with known bits $b_1 \cdots b_i$, it is computationally infeasible to calculate $b_{i+1} \cdots b_{i+n}$, with a probability significantly greater than one-half. [PP10]

The next-bit test is based on the concept of unpredictability. It is a "complete statistical test" in that if a pseudorandom sequence passes this test, it will definitely pass all other randomness quality tests [Shelat]. A sequence passes the next-bit test if an attacker —at most— can predict the next generated bit with a probability insignificantly greater than one-half. This test assumes that an attacker has knowledge of some or all of the previously generated bits. The probability index is a derivation of a truly random sequence probability distribution. The formal definition of the next-bit test is as follows:

Let pseudorandom sequence X be on $\{0, 1\}^m$, time $t \in \mathbb{N}$ and ε be some arbitrarily small number. X is *(t,ε)* next-bit unpredictable if for every probabilistic algorithm A running in t and for all i ,

$$Pr[A(x_1 x_2 \cdots x_i) = x_{i+1}] \leq \frac{1}{2} + \varepsilon.$$

Note: The next-bit test is abstract. No tangible procedure currently exists. It is also evident the test parallels the principle of unpredictability. The property of unpredictability assures whether or not a PRNG is safe for cryptographic applications.

All pseudorandom quantities do not possess the same standard of quality. Therefore, restrictions must be made on where they may substitute random sequences. It is fair to say, that cryptographically secure pseudorandom number generators may be used wherever random quantities are needed. They are computationally infeasible like true random-producing objects. However, non-cryptographic generators are limited in use. This is attributed to the existence of distinguishers and their lack of unpredictability. Even though cryptographically secure generators can be used anywhere, it need not be used everywhere. They are only needed when unpredictability is the main concern of the requesting application. In cases where unpredictability is not a necessity, a standard generator will suffice.

3.3 Formalized Components for CSPRNGs

3.3.1 RFC4086

The Internet Engineering Task Force's RFC4086 is a publication that outlines and is titled, "Randomness Requirements for Security." The article serves as a guideline for constructing random generators resistant to cryptanalytic attacks and safe to use in cryptosystems. RFC4086 is a revamped version of the obsolete RFC1750. Unlike its predecessor, RFC4086 provides additional information on various entropy definitions and methods (Renyi entropy, minimum entropy and entropy pool techniques); mixing functions using S-boxes; as well as a listing of additional sources of randomness.

The authors were motivated to write this document because of several issues that cause debilitating affects to cryptographic software. The affects leverage the probability of guessing or determining data to be more advantageous to an attacker. One set of attributes is the absence of truly random or unpredictable resources. Cryptographic software (like SSH, IPSEC and PGP) typically selects random quantities that only adhere to traditional statistical requirements; or are obtained from a small set of random values or easily calculable resources. Another cause for the authors to create this document was the difficulty to construct a generator that is compatible with a diverse selection of hardware.

Some possible solutions described in the text were hardware-based entropy sources, de-skewing methods and endorsed mixing functions. Hardware-based entropy sources suggested for seeding a PRNG are thermal noise, radioactive decay, ring oscillators, audio and video input device, spinning disks as well as crystal time sources. The authors further explained that in order for these entropy sources to be available, computer vendors must first integrate them in their products. De-skewing relates to transforming a random output (of bits) such that distribution of zeroes and ones are symmetrical. Some of the methods suggested for de-skewing are using stream parity, transition mappings (a technique introduced by John von Neumann) and compression. Lastly, mixing is used to help compensate for the lack of hardware-based entropy sources available in a computing system. An entropy preserving algorithm is used to mix entropy quantities drawn from unrelated sources and produce a strong seed. A robust mixing algorithm is

defined as a process "that combines inputs and produces an output in which each output bit is a different complex non-linear function of all the input bits" [RFC4086].

The article also claims that cryptographically secure random outputs are first built on a strong seed; subsequent steps following seed generation must also be cryptographically secure and the complete state of the generator must be kept private. The authors stated that the correct technique for producing cryptographically secure random sequences is founded on these two principles.

3.3.2 FIPS PUB 140-2 (Security Requirements for Cryptographic Modules)

This publication provides a standard for cryptosystems used by U.S. Federal organizations or affiliated organizations working with sensitive federal data. There are eleven sections with requirements differing over four levels of security. The sections include "cryptographic module specification; cryptographic module ports and interfaces; roles, services, and authentication; finite state model; physical security; operational environment; cryptographic key management; electromagnetic interference/electromagnetic compatibility (EMI/EMC); self-tests; design assurance; and mitigation of other attacks" [FIPS 01].

One of the most significant security area described, pertaining to CSPRNGs, is section 4.9 Self-Tests. CSPRNGs are subjected to two tests: power-up self-tests and conditional self-tests. Power up-self tests are system-automatic assessments carried out whenever the generator is initialized. There are few types of power-up tests available for CSPRNGs but the "cryptographic algorithm test" is the most relevant. This test confirms the correctness of a generating algorithm. Specific sequences, whose outputs are known, are used to seed the generator. Then, the random value produced during the assessment is compared to the expected output. Like power up-self tests, conditional self-tests also include various assessments. "Continuous random generator test" is the most appropriate for CSPRNGs. The test checks if the first random quantity produced is equal to the next generated quantity. If the two are equal, the generator failed the test.

3.4 Proposed Security Models for CSPRNGs

3.4.1 Barak-Halevi Model

Barak and Halevi in 2005 introduced a new security criterion called robustness and stated that this is a key property for any cryptographically suitable PRNG [BH05]. Once robustness is present, resiliency, forward security and backward security are also present. Principally, robustness gauges the security of a PRNG after a state compromise;[3] Robustness indicates if a generator is capable of upholding the three security properties aforementioned. Resiliency, forward security and backward security were formulated from research describing various advantages an adversary may have when trying to attack a PRNG. The advantages are as follows: the adversary has access to the random output; the adversary can influence the entropy source; and the adversary can manipulate the generator's internal state. The attacker may have only one or a combination of the advantages noted. A game playing framework based on [BR06] and

[3] State compromise refers to the corruption or manipulation of the internal state of a generator by an external source.

[DPR+13] will be used to gain a better understanding of robustness by illustrating its security through forward secrecy, backward secrecy and resiliency games.

Games are programs with three types of methods: initialization, oracles (a set of individual methods) and finalization. The oracles consist of the following methods: **refresh** which refreshes the current state of the PRNG, **getNext** which gets the next random quantity from the PRNG, **getState** which gets information on the current state of the PRNG and **setState** which sets the state to a value chosen by the adversary. Calls are made to the oracles by the adversary, which is also a program.

The game begins by invoking the initialization method. The outputs are passed to the adversary. The adversary takes these outputs and makes calls to an arbitrary number of oracles. The adversary then returns a final value to the finalization method. Lastly, the finalization method returns a value. This is the output of the game and marks the end as well.

Resilience is an attacker's inability to predict future outputs even if the entropy source—whose purpose is to update the internal state of the generator—is compromised by input by the attacker. The adversary has no information on the internal state of the generator. In a resiliency game, the adversary makes no calls to the **getState** or **setState** methods.

In forward secrecy, an adversary cannot predict past outputs even if he can influence the internal state of the PRNG. The adversary is restricted from making calls to the **setState** method. It can only invoke the **getState** method once. This call is the final call made by the adversary. [DPR+13]

Reversely, backward secrecy is the inability to predict future outputs after an internal state compromise. During a backward secrecy game, the adversary does not invoke the **getState** method but makes only one call to the **setState** method. This is the initial oracle call made by the adversary. The output returned by the finalization method should demonstrate full recovery from the attack. That is to say, the output should be unpredictable after this state compromise.

Robustness entails all the aforementioned security criteria. Principally, robustness gauges the security of a PRNG after a state compromise; it indicates if a generator was able to uphold the standards of resilience, forward secrecy and backward secrecy. In a robustness game, an adversary can make calls to all of the oracles.

Note: In all games, the adversary may make calls to the **refresh** and **getNext** methods.

3.4.2 DPRVW Model

Dodis, Pointcheval, Ruhault, Vergnaud and Wichs extend the security model proposed by Barak and Halevi by including a new security property informally called *entropy accumulation* [DPR+13]. According to the researchers, this property is the critical concept used to strengthen the robustness notion proposed by Barak and Halevi. Entropy accumulation describes how entropy, the measure of disorder (randomness), should be collected over a period of time while a PRNG is in operation. In common practice, PRNGs try to accumulate high entropy in one instance— when the system's state is refreshed. *Entropy accumulation* slowly collects entropy

throughout the entire operation of the PRNG and uses this slowly accumulated entropy to refresh the state. The new property satisfies the authors' requirement that a good pseudo–random number generator is one that can ultimately recover after a state compromise.

The model considers two types of adversaries: one, the conventional adversary better known as a distinguisher and two, a distribution sampler. The latter adversary aids a distinguisher by feeding the generator with high entropy inputs.

The PRNG construction proposed in [DPR+13] is defined as an algorithm which consists of three explicit functions: **setup()**, which outputs a seed, **refresh(*S*, *I*)**, which updates the state, and **next(*S*)**, which make a single call to **refresh** and outputs a pseudorandom quantity.

Before we look at the functions proposed, let us define the following variables:

- $m, n, p \in \mathbb{N}$, m is the input length, n is the state length and p is the output length.
- X represents a distribution on $\{0,1\}^m$ and X' is some derivative of the distribution.
- The generator's current state $S \in \{0,1\}^m$.
- entropy $I \in \{0,1\}^m$.
- pseudorandom quantity $R \in \{0,1\}^p$.
- U denotes a uniform distribution.

All operations performed in the functions below are in a field of 2^n elements. Therefore, addition represents XOR operation and multiplication represents the AND operation.

Let $\mathcal{G}$ be a pseudorandom number generator where $m < n$, then

$$\mathcal{G}: \{0,1\}^m \rightarrow \{0,1\}^{n+p}$$

setup()
//Output: seed (X, X')
$(X, X') \leftarrow \{0,1\}^{2n}$

It is important to note that in this model, the seed is a calculated entity. Additionally, it is not hidden from an adversary.

refresh(S, I)
//Input: seed, S, I
//Output: a new state S'
$S' \leftarrow S \cdot X + I$

next(S)
//Input: seed (X, X'),
//Output: a new state S',
$U = [X' \cdot S]_1^m$
$(S, R) = \mathcal{G}(U)$

In the next procedure, step $U = [X' \cdot S]_1^m$, employs an on-line extractor, which is an extractor that performs in running time. The extractor is called by pseudorandom number generator G and operates conditionally. This step is bypassed whenever the last call made by the generator was the **next** procedure. The purpose of the extractor is to defray the effects of a chosen input attack. Here, the extractor is a compounded hash function. According to the authors, such function ensures the quantity obtained reflects uniformity or true randomness.

We will now discuss the mechanism of the online extractor, a crucial component of the DPRVW generator. From the **refresh** function, the output S′ can also be expressed as the following:

$$S' = S \cdot X^j + I_{j-1} \cdot X^{j-1} + I_{j-2} \cdot X^{j-2} + \cdots + I_1 \cdot X + I_0,$$

where j is the number of distribution samples.

In actuality the **refresh** function is a hash function that computes the following:

$$\text{Hash}_X(I) := \sum_{j=0}^{i-1} I_i \cdot X^i.$$

To ensure security, a second hash function is introduced. Let n be the state length and J be some input $\epsilon \{0, 1\}^n$, so

$$\text{Hash}_Y(J) := [X' \cdot J]_1^m.$$

The two functions are combined in the following hash function to extract high quality randomness during the **next** procedure.

$$\text{Hash}_{X,Y}(I) := \left[X' \cdot \sum_{i=0}^{j-1} I_i \cdot X^i \right]_1^m.$$

This compound hash function is an online (k,ε)extractor if

$$k \geq m + 2 \log\left(\frac{1}{\varepsilon}\right) + 1, n \geq m + \log(j) + 1.^4$$

3.5 Some CSPRNG Implementations

3.5.1 Fortuna

"After analyzing existing PRNGs and breaking [their] share of them, [Bruce Schneier and Niels Ferguson] wanted to build something secure."[5] "Fortuna: Cryptographically Secure Pseudo-Random Number Generation in Software and Hardware" discusses Fortuna, a CSPRNG created by cryptographers Niel Ferguson and Bruce Schneider [FN03]. Fortuna is a PRNG designed to

[4] The proof can be found eloquently written in [DPR+13].

[5] https://www.schneier.com/academic/fortuna/

be resilient from known cryptographic attacks. The generator possesses properties that are considered acceptable for cryptographic application. Aside from resiliency, it is computationally impossible to predict the next bit of a known sequence (backward secrecy) and previous sequences cannot be determined even if part or all of the internal state is known (forward secrecy). The document also discussed the generator's entropy sources, a software implementation of Fortuna in C++, a hardware implementation as well as an analysis of the generators.

Fortuna is an improvement to Yarrow, a generator that has been incorporated in systems like iOs, Mac OS X and FreeBSD[6] [7]. What makes Fortuna different from Yarrow is that it eliminates the use of entropy estimators— an entity whose definition is generally accepted as impractical. Fortuna consists of three parts: a generator, an entropy accumulator and a seed file manager. The generator takes a seed of fixed size and yields pseudorandom numbers of various lengths. The structure of the generator includes a counter mode block cipher, which includes a key. The generating algorithm consists of four main procedures: **InitializeGenerator**, **Reseed**, **GenerateBlocks** and **PseudoRandomData**. The first procedure sets the cipher key and counter to zero to indicate that the generator has not been seeded. The second procedure refreshes the internal state of the generator with an arbitrary bit sequence. A hash function is used to mix the arbitrary sequence and the current key. The **GenerateBlocks** function produces blocks of random quantities. It is a private function in that it can only be called by the generator. Contrarily, the fourth procedure allows an external entity to request random output. Only 2^{20} bytes of data can be given per request. Entropy is collected from various sources in the system and stored in pools by the accumulator; it also seeds and reseeds the generator from time to time. Up to 256 sources of entropy may be used by the generator. Three common ones are discussed in the next paragraph. Furthermore, even if some of the entropy sources are influenced by an adversary, the generator is resilient to this attack because of the multiple entropy pools available. The last component, the seed file manager, ensures the production of randomness even under the circumstances that the system has recently been booted.

The software implementation discussed in this article used three entropy sources: mouse movement, keystroke timing and noise from the PC's soundcard. These sources were included in the construction because of their accessibility. With mouse movement, the position of the cursor is consistently recorded and added to the entropy pool. The authors stated that mouse movement is truly random because it is derived from unsystematic user behavior and computationally, "the least significant denominations of its position cannot be guessed" [MCC+06]. Entropy gathering in keystroke timing consists of recording the times– in milliseconds– between keystrokes. The least significant quantities of the recorded times are added to the entropy pool. The third source was included in the implementation to ensure a constant source of entropy. Mouse movement and keystroke timing both depend on user behavior; soundcard noise does not. Also, Fortuna is resilient to attacks if some– not all– of the entropy sources are compromised. Soundcard noise is more difficult to influence in contrast to the other two sources. It helps to prevent the cryptosystem from collapse.

---

[6] https://www.apple.com/br/ipad/business/docs/iOS_Security_Oct12.pdf
[7] https://svnweb.freebsd.org/base?view=revision&revision=284959

3.5.2 /dev/random

The Linux kernel has two PRNGs, /dev/random and /dev/urandom. The former is the only one used to provide pseudorandom outputs for cryptographic applications running on the OS. This is because /dev/random is a blocking PRNG and /dev/urandom is a non-blocking PRNG. Blocking PRNGs will not respond to a call until the minimum entropy level has been reached; all operations are delayed. Non-blocking PRNGs will always operate regardless of the amount of entropy present in the system. The latter is deemed unsuitable for use in cryptosystems since it yields outputs even in cases of insufficient entropy.

The /dev/random construction contains two entropy pools: an input pool for collecting entropy from external sources; and an output pool for generating pseudorandom sequences. The blocking component works via an entropy estimation function. The algorithm approximates the entropy of the input values used to update the input pool. In actuality, an input value is not directly used to measure entropy but its timing. The entropy estimation function calculates the time differences between the current input value and the immediate preceding value. This step is compounded two additional times. Then, the minimum of the absolute value of all three differences is computed. Next, the floor of half the minimum value undergoes modulo 2^{12} operation. Lastly, a logarithmic function applied to last value resulting in the 12 bit entropy value for the current input.

Let time t be on $\{0, 1\}^{32}$. The entropy estimate of input m, H_m, is computed by doing the following:

$$\delta_m^1 = t_m - t_{m-1}$$
$$\delta_m^2 = \delta_m - \delta_{m-1}$$
$$\delta_m^3 = \delta_m^2 - \delta_{m-1}^2$$
$$\delta_{\mathrm{m}} = \min(|\delta_m|, |\delta_m^2|, |\delta_m^3|)$$
$$H_m = \log_2\left(\left|\frac{\delta_{\mathrm{m}}}{2}\right| \bmod 2^{12}\right)$$

[Roe05][DPR+13][GPR06]

In [DPR+13], the authors stated that it is impractical and theoretically impossible to estimate entropy. This immediately leads to notion that the entropy estimation procedure in /dev/random is inaccurate and an unreliable security mechanism. The authors also successfully attacked the PRNG by exploiting the weakness of its entropy estimation function. More information about the attack can be found in the paper.

3.6 Lightweight Cryptographic PRNGs

3.6.1 Lightweight Cryptography

As technology advances, there are a growing number of small, compact computing devices capable of exchanging information over new types of network. With the increasing popularity of these devices, concerns have risen about the security or privacy of communication flowing to and from these "constrained devices." Any device that has limited processing power, memory storage capabilities, and power resources are labeled as constrained devices. Internet of things, sensor networks, smart object networks, and cyber physical systems are prime examples of networks that host constrained devices.

Lightweight cryptography is a cryptographic protocol or algorithm intended for usage in constrained device networks. The cryptographic algorithms/implementations discussed prior and other traditional algorithms are more suited to operate in desktop/server environments. Research efforts have been concentrated on maximizing and maintaining the balance between security, performance and resource consumption specific to standard computing networks.

Constrained devices, typically, do not possess the proper resources to employ traditional cryptographic algorithms. There are some cases where traditional algorithms can be employed but it is also accompanied by significant performance reduction. Performance encompasses power and energy consumption as well as latency and throughput. Lightweight cryptography is intended to provide a solution for the performance-security tradeoff problem that exists for compact devices.

Lightweight cryptosystems may be implemented on either hardware platforms or in software applications. The time and space resources required to execute the program defines if an algorithm is lightweight. Hardware-based algorithms and software-based algorithms use different metrics to measure time and space resource consumption. For hardware, time consumption is measured by the throughput in bits per second given a computation frequency and latency. Required space resources are determined by the number of logical gates required for execution. For software, time consumption is the number of clock cycles necessary to process data of one-byte lengths. Space costs are "measured by the number of registers and the number of bytes of RAM and ROM need" [MBT+17]. Furthermore, the amount of resources a lightweight cryptographic algorithm use relies on the size of the constrained and other design objectives imposed by the device [MBT+17].

Note: Even though there are guidelines available for hardware and software implementations, most lightweight cryptosystems are only implemented in hardware. This can be attributed to the limitations of the computing devices as well as the efficiency of hardware implementations.

Currently, large efforts are being made to construct efficient lightweight cryptographic algorithms to be implemented in constrained environments. NIST has also been active in this effort and is moving towards establishing a lightweight cryptography standard. [MBT+17Later we will discuss a lightweight cryptographic algorithm, Trivium, which is one of the earlier approaches catering to this subfield in cryptography.

3.6.2 J3Gen

J3Gen is a lightweight PRNG explicitly designed for RFID tags. For this reason, J3Gen is confined to hardware implementation. The structure of the generator entails four components: LFSR, Polynomial Selector, RNG and Decoding Logic. The LFSR component describes the structure of the generating function which is a linear feedback register. To counter the linearity effects of an LFSR, the authors designed the generator to operate from a set of feedback functions rather than one. The Polynomial Selector chooses the function the generator employs.

Feedback functions are stored in a circular array. The Polynomial Selector moves along the array by one or two functions depending on the bit generated by the RNG. A zero bit denotes a single function shift and a one-bit signifies a two function shift. The RNG is based on thermal noise, specifically, an oscillator-based high frequency sampler. The generated bit is not received directly by the Polynomial Selector but passed by the Decoding Logic component. [MGH13]

3.6.3 PRESENT

PRESENT is an ultra-lightweight block cipher created to operate in constrained environments. The authors imposed a hardware implementation-only restriction on the construction. The core design principles for PRESENT are (1) a block cipher that is secure and (2) a block cipher that performs well in hardware. The authors of the block cipher also intended for the cipher to function as a PRNG in environments where it is more efficient to have a built-in generator than stored random values.

The PRNG implementation of PRESENT is accomplished by operating the cipher in OFB mode. The internal state of this generator is 64 bit long. It may be initialized with an 80-bit seed or a 128-bit seed. The main algorithm fashions an SP-network of 31 rounds. More details about the procedures can be found in [Bog+07].

Chapter 4

SecureRandom

In Java, an instance of java.util.Random constructs a pseudorandom number generator and produces sequences of pseudorandom numbers.[8] The PRNG is modeled after the linear congruential method and uses a 48-bit length initial value. A Random class object can produce up to 32-bits of pseudorandom numbers per call to the generating function. [OracleRa]

As stated on Oracle's website, "Instances of java.util.Random are not cryptographically secure."[9] This statement is backed by a number of research presenting attacks on the linear congruential method. An adversary is capable of guessing the subsequent or previously generated numbers in the event of a security breach. A direct subclass of the Random class, SecureRandom is the recommended CSPRNG for any Java program that includes a cryptographic algorithm. SecureRandom is featured in the Java Cryptographic Service Providers package.[10] Depending on the source of randomness, implementations of SecureRandom can be in the form of a PRNG or RNG. [OracleSR]

SecureRandom is built on top of an underlying PRNG and uses methods inherited from the Java Random class. This leaves an implementation flexible to choose the base PRNG. SecureRandom relies on an external source to provide a 160-bit seed for initialization. The seed provider is dependent on the operating system the generator is running on or a user specified PRNG. In the case of a Solaris/Linux environment, /dev/urandom, the system's native PRNG, supplies the seed. On Windows, a SHA1PRNG provides the seed by default.

For instances when the underlying PRNG is not specified, the Sun Microsystem implementation is applied by default. Some implementations like ApacheHarmony and BouncyCastle include additional methods to ensure the security of their respective generators. The OpenJDK version is the official implementation of SecureRandom and the version chosen for discussion.

4.1 Structure

The internal state of SecureRandom is 160 bit in length. The main phases of the generator are the initialization phase, seeding phase and generation/update phase. The authors of SecureRandom OpenJDK consider the generator to be a "self-seeding" CSPRNG. In fact, SecureRandom is not a truly self-contained construction and relies on an external class, SeedGenerator, to aid in supplying a seed for the algorithm.

4.2 Algorithm

Since the OpenJDK version of SecureRandom is a part of the Java Library, only the following is needed to create an instantiation of the class:

---

[8] java.util.Random is capable of returning numbers other than bits.
[9] https://docs.oracle.com/javase/8/docs/api/java/util/Random.html
[10] This provides security compliant to NIST document FIPS 140-2.

```
SecureRandom sr = new SecureRandom();
byte[] bytes = new byte[size];
sr.nextBytes[bytes];
```

The initialization process for SecureRandom begins by creating an instance of MessageDigest SHA1. The PRNG is not seeded until a call is made to the **engineNextBytes** method.

The seeding process begins once the generating procedure is invoked. At this step, the algorithm instantiates an inner class called SeederHolder. SeederHolder creates another instance of SecureRandom sr_{inner} but with a specified seed, s_i, as input. Seed s_i is retrieved from the SeedGenerator class. The class collects entropy from the system then executes a SHA1 hash on the collected data. Some entropy sources include the current time in milliseconds; runtime memory data; system property information like the operating system name, architecture and version as well as property information about the Java Virtual Machine and Java Runtime Environment. The hash of the entropy sources is assigned to s_i. The state of sr_{inner} is assigned the hash of s_i. (This is the first state of the PRNG and simply, a double hash on the system entropy.)

SeederHolder makes a second call to SeedGenerator to provide 160 bits of pseudorandomness. The hash on this quantity and the first state is assigned to the second state of sr_{inner}. Finally, sr is seeded with s, the hash of the second state.

$$s = Hash_Z\big(Hash_Y(Hash_X(s_i) + J)\big), \text{ where } J \text{ is some pseudorandom input.}$$

The **engineNextBytes** method is responsible for generating pseudorandom quantities. SecureRandom generating process begins by updating the digest with the current state. Then, it performs a hash operation on the state. The output is a 20-byte long sequence. Let s_i denote the i^{th} state bit, the following formula is used to update the state:

$$[(s_1, s_2 \cdots s_{160}) + Hash(s_1, s_2 \cdots s_{160}) + 1] \, mod \, 2^{160},$$

Here is the OpenJDK code snippet focusing on the main generating procedure:

```
while (index < result.length) {
    digest.update(state);
    output = digest.digest();
    updateState(state, output);

    todo = (result.length - index) > DIGEST_SIZE ?
        DIGEST_SIZE : result.length - index;

    for (int i = 0; i < todo; i++) {
        result[index++] = output[i];
        output[i] = 0;
    }
    remCount += todo;
}
```

The variable `result` denotes the byte array that holds the pseudorandom sequence. With each state, SecureRandom produces at most 20 bytes at a time. The while-loop concatenates each 20 byte sequence and does not break until the desired length is attained.

4.3 Output

SecureRandom output can be simplified to:

$$(z_1, z_2 \cdots z_{160}) \leftarrow Hash(s_1, s_2 \cdots s_{160}).$$

The security lies in the reliability of the SHA message digest. Additionally, two consecutive internal states may not be identical. This will ensure a change in consecutive outputs.

Chapter 5

Trivium

A stream cipher is a symmetric cipher that encrypts bits individually. Stream ciphers can be synchronous or asynchronous. Synchronous types are ciphers whose keystreams only depend on the key. In contrast, asynchronous types are stream ciphers whose keystreams are dependent on the ciphertext. The keystream, the output of a stream cipher, may be used as a source for pseudorandom bits.

Trivium is a synchronous stream cipher created by Christophe De Canniere and Bart Preneel. It was submitted to the eSTREAM competition and selected for the eSTREAM portfolio of lightweight stream ciphers for hardware application. Despite its intent for hardware, Trivium is still efficient in software-based environments. Additionally, it has been designated by International Organization for Standardization (ISO) as a keystream generator for lightweight stream ciphers.

Trivium can also be described as a bit-oriented stream cipher conducting operations at the bit level. The internal state of the cipher consists of three registers totaling to 288 bits. The first register holds 93 state bits, the second holds 84 state bits and the last register holds 111 state bits. The algorithmic component is broken down into two phases, the setup phase and the generation phase (which is also responsible for updating the internal state of the cipher). Trivium takes in a key and IV of 80 bits each and guarantees to generate up to 2^{64} keystream bits. [DP05]

5.1 Structure

When creating Trivium, the authors had two mandatory specifications the construction must contain. First, the structure must generate seemingly uncorrelated keystreams. Second, the construction must also be efficient such that there is a high throughput of generated keystream bits per cycle per logic gate. The authors referenced the operations of block ciphers as a solution to their specifications. In comparison to stream ciphers, block ciphers are more developed. Many techniques have been uncovered to bolster the efficiency of block ciphers to operate speedily and low space consumption. Additionally, the security of a block cipher is well researched and understood. The internal structure of a secure block ciphers have been well defined.

Block ciphers achieve linear independence amongst input and output values by alternating between two operations. Each block is first partitioned into smaller data units and transformed using distinct substitution boxes (S-boxes). The second operation entails combining the transformed data units differently to reconstruct the block. The two operations foster confusion and diffusion within each block. In block cipher algorithms, this dual procedure is better known as a substitution-permutation network (SP-network).

Block ciphers can function like stream ciphers, generating keystreams, if they are operated in Cipher Feedback (CFB) mode, Output Feedback (OFB) mode and most popularly, Counter (CTR) mode. CFB takes an initialization vector (IV) and passes it through the cipher. The plaintext is then XORed with the output. The ciphertext generated is then used as input to the successive round of encryption. This process may be repeated to obtain a keystream of n-length. OFB mode

starts with an IV and passes it to the cipher as input. The immediate output obtained is passed to the successive round as an input. The process is reiterated until the desire keystream size is achieved. CTR mode uses a nonce (which is equivalent to the IV used in the aforementioned modes) as the initial block. A counter value is used as cipher input. The successive round uses (counter + 1) as input. [Dwo01]

Remember, block ciphers are cryptographic algorithms that handle data as blocks of information. Stream ciphers are algorithms that handle data as a stream. We will now present how Trivium has taken the operations of block ciphers and translated them to correspond with the structure of a lightweight stream cipher.

In the descriptions of the modes of operation, we saw that encryption is done as a succession of rounds. At each round, the SP-network provides diffusion via a linear diffusion function and confusion employing a set of S-boxes. In Trivium, the concept of rounds in block ciphers is translated as registers. The cipher is partitioned into three registers. As for S-boxes, they are replaced with a special structure. The set of S-boxes that are found at each round is reduced to a single 1 x 1 bit block for each register. The 1x1 bit S-box transforms a bit via an XOR operation with a bit residing in another register. The linear diffusion function is a linear filter; a technique that transforms a bit using linear combinations of neighboring bits. Furthermore, Trivium specifically mirrors the mechanism of a block cipher in OFB mode. Outputs from one register are fed as input to another register.

5.2 Algorithm

Trivium requires an 80-bit key and 80-bit initialization vector for set up. Initialization begins with the key being copied to the first register. After copying the key to the first 80 slots, the remaining state bits are set to zero. The initialization vector is then written to the second shift register. The rightmost four bits in this register are set to zero. The last register has all its bits set to zero except for the last three bits; they are set to one. The internal state is refreshed 1152 times to ensure that all bits are influenced by the key and the IV. The pseudocode is given as the following:

$[s_1, s_2, \cdots, s_{93}] \leftarrow [K_1, \cdots, K_{80}, 0, \cdots, 0]$
$[s_{94}, s_{95}, \cdots, s_{177}] \leftarrow [IV_1, \cdots, IV_{80}, 0, \cdots, 0]$
$[s_{178}, s_{179}, \cdots, s_{288}] \leftarrow [0, \cdots, 0, 1, 1, 1]$

FOR i = 1 to 1152

$t_1 \leftarrow s_{66} + s_{91} \cdot s_{92} + s_{93} + s_{171}$
$t_2 \leftarrow s_{162} + s_{175} \cdot s_{176} + s_{177} + s_{264}$
$t_3 \leftarrow s_{243} + s_{286} \cdot s_{287} + s_{288} + s_{69}$
$[s_1, s_2 \cdots s_{93}] \leftarrow [t_3, s_1 \cdots s_{92}]$
$[s_{94}, s_{95} \cdots s_{177}] \leftarrow [t_1, s_{94} \cdots s_{176}]$
$[s_{178}, s_{179} \cdots s_{288}] \leftarrow [t_2, s_{178} \cdots s_{287}]$

END FOR

The Trivium generation process actually begins by performing an exclusive or operation on two specific bits from each register. The resulting three bits collectively undergo another exclusive or operation. The result from this last step is a single bit that is added to the keystream. The pseudocode is below:

//m = number of pseudorandom bits requested
FOR $i = 1$ to m

$t_1 \leftarrow s_{66} + s_{93}$
$t_2 \leftarrow s_{162} + s_{177}$
$t_3 \leftarrow s_{243} + s_{288}$
$z_i \leftarrow t_1 + t_2 + t_3$

//Trivium updates by doing the following:
$t_1 \leftarrow t_1 + s_{91} \cdot s_{92} + s_{171}$
$t_2 \leftarrow t_2 + s_{175} \cdot s_{176} + s_{264}$
$t_3 \leftarrow t_3 + s_{286} \cdot s_{287} + s_{69}$
$[s_1, s_2 \cdots s_{93}] \leftarrow [t_3, s_1 \cdots s_{92}]$
$[s_{94}, s_{95} \cdots s_{177}] \leftarrow [t_1, s_{94} \cdots s_{176}]$
$[s_{178}, s_{179} \cdots s_{288}] \leftarrow [t_2, s_{178} \cdots s_{287}]$
END FOR

Trivium produces only a single bit at a time. This entire process is reiterated until the desired length is reached.

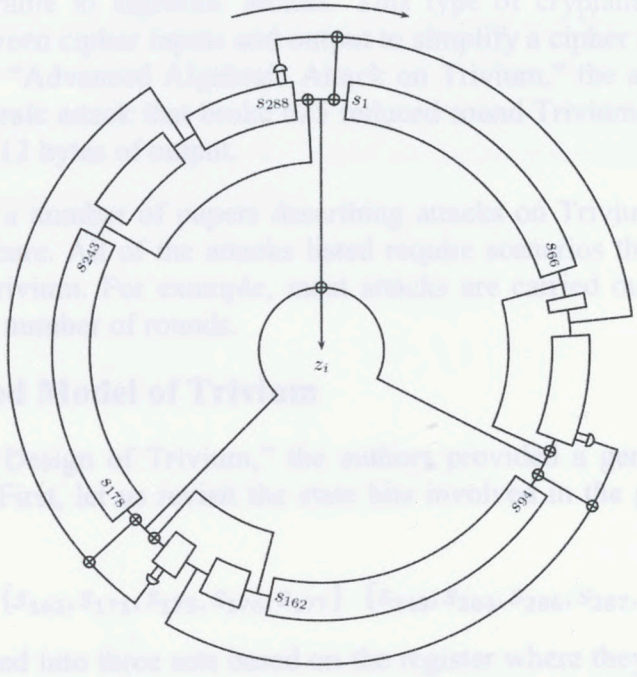

Figure 2. Structure of Trivium

5.3 Output

The pseudorandom output can be simplified to:

$$z_i \leftarrow s_{66} + s_{93} + s_{162} + s_{177} + s_{243} + s_{288}.$$

The unpredictability of the z_i is dependent on the constant rotation of the state bits and transformation of some bits with each state update. An attacker must be aware of the internal state to accurately predict the next bit. This is quite difficult given that each Trivium state is constructed to be linearly independent.

5.4 Scholarly Attacks

In Dinur and Shamir's "Cube Attacks on Tweakable Black Box Polynomials," the authors introduced a new class of attacks to which stream ciphers and other cryptographic schemes exhibit vulnerability. Many cryptographic systems can be defined by a system of polynomial equations. The cube attack is a method that solves such equations by computing a sum over the output bits generated by a set of chosen IVs. Dinur and Shamir were able to recover the key of a 672 reduced round Trivium variant in 2^{19} bit operations. Before the cube attack, the best attack on the same variant required 2^{55} bit operations. The cube attack is efficient in recovering the full key up to 767 rounds of Trivium.

Another paper, "Cube Attacks on Trivium," furthered explored this cryptanalytic method by combining it with other known technique. Pierre-Alain Fouque and Thomas Vannet were able to recover the full key of a 799 reduced round Trivium variant in 2^{62} bit operations.

Trivium is also vulnerable to algebraic attacks. This type of cryptanalytic attack establishes algebraic relations between cipher inputs and output to simplify a cipher to a system of equations or to recover the key. "Advanced Algebraic Attack on Trivium," the authors, Quedenfeld and Wolf, showed an algebraic attack that broke 625 reduced round Trivium variant. The attack was successful using only 512 bytes of output.

Even though there are a number of papers describing attacks on Trivium, please keep in mind that Trivium is still secure. All of the attacks listed require scenarios that are not typical to the normal operation of Trivium. For example, most attacks are carried out on reduced rounds of Trivium versus the full number of rounds.

5.5 A Generalized Model of Trivium

In the paper, "On the Design of Trivium," the authors provided a generalization of the main algorithm in Trivium. First, let us revisit the state bits involved in the generating and updating processes:

$$\{s_{66}, s_{69}, s_{91}, s_{92}, s_{93}\} \ \{s_{162}, s_{171}, s_{175}, s_{176}, s_{177}\} \ \{s_{243}, s_{264}, s_{286}, s_{287}, s_{288}\}$$

The state bits are divided into three sets based on the register where they reside. Taking a closer look, there are a few patterns which emerge across the sets. First, in each set the first, second and fifth state bit index numbers are all multiples of three. Also, the third and fourth and fifth state

bits in each set are consecutive bits. The fifth state bits are the last state bit for their respective registers. Also, the first and fifth bits are a part of the special S-box construction in Trivium. If we treat the indices as variables we get the following:

$$\{s_{3m_1}, s_{3m_2}, s_{3n_1-2}, s_{3n_1-1}, s_{3n_1}\}\ \{s_{3m_3}, s_{3m_4}, s_{3n_2-2}, s_{3n_2-1}, s_{3n_2}\}\ \{s_{3m_5}, s_{3m_6}, s_{3n_3-2}, s_{3n_3-1}, s_{3n_3}\},$$

where $m_1 < m_2 < n_1 < m_3 < m_4 < n_2 < m_5 < m_6 < n_3$. The variable n_i expresses the upper most bit in a register. Index i corresponds with the register location of a state bit.

The generalized version of Trivium's main procedures is as follows:

$$t_1 \leftarrow s_{3m_1} + s_{3n_1}$$
$$t_2 \leftarrow s_{3m_3} + s_{3n_2}$$
$$t_3 \leftarrow s_{3m_5} + s_{3n_3}$$
$$z_i \leftarrow t_1 + t_2 + t_3$$

$$t_1 \leftarrow t_1 + s_{3n_1-2} \cdot s_{3n_1-1} + s_{3m_4}$$
$$t_2 \leftarrow t_2 + s_{3n_2-2} \cdot s_{3n_2-1} + s_{3m_6}$$
$$t_3 \leftarrow t_3 + s_{3n_3-2} \cdot s_{3n_3-1} + s_{3m_2}$$
$$[s_1, s_2 \cdots s_{3n_1}] \leftarrow [t_3, s_1, \cdots, s_{3n_1-1}]$$
$$[s_{3n_1+1}, s_{3n_1+2} \cdots s_{3n_2}] \leftarrow [t_1, s_{3n_1+1}, \cdots, s_{3n_2-1}]$$
$$[s_{3n_2+1}, s_{3n_2+2} \cdots s_{3n_3}] \leftarrow [t_2, s_{3n_2+1}, \cdots, s_{3n_3-1}]$$

From this result, the authors were able to decompose Trivium into Univium, a 1-round Trivium based cipher, and Bivium, a 2-round Trivium based cipher. The upper bound of Univium is state bit n_1 and for Bivium, it is n_2. Furthermore, they proposed the k-round Trivium based cipher that extends Trivium to some $k \in \mathbb{N}$ and selects the state bits based on a k-order primitive polynomial. We will discuss this further in the next chapter.

Chapter 6

Quadrivium

Quadrivium is a pseudorandom number generator designed with a software implementation in mind. The structure is primarily modeled after Trivium but coalesce findings in [TCL09]; the definition of primitive polynomials; and feedback functions found in linear feedback shift registers (LFSRs).

6.1 Design

Quadrivium is a 384 bit state PRNG. The generator is partitioned into four registers of 98-bit, 97-bit, 95-bit and 94-bit length. It requires a total of 160 random bits for initialization.

To understand the design principles of Quadrivium, first, we will review some standard definitions relating to polynomials.

Definition (polynomial) A polynomial *p(x)* is a mathematical expression consisting of a sum of terms where each term includes x raised to a non-negative integer power and multiplied by a coefficient. It can be written as

$$p(x) := \sum_{i=0}^{n} a_i x^i,$$

where the variable a_i denotes the coefficient of x_i and a_n is different from zero. The value of n is the degree of *p(x)*. For this discussion, we restrict all *p(x)* to polynomials in $GF(2)$. Therefore, all coefficients are either zero or one.

A polynomial *p(x)* is said to be trivial if the degree of *p(x)* is $-\infty$, indicating a zero polynomial, or 0, a constant polynomial; otherwise, it is nontrivial.

Definition (irreducible polynomial) A polynomial *p(x)* is irreducible if it cannot be factored into two or more non-trivial polynomials.

Definition (primitive polynomial) A primitive polynomial is an irreducible polynomial with degree $n > 0$ and a polynomial order (or period) of 2^n-1.

Definition (Linear Feedback Shift Register) An LFSR is an object that employs the function $f: \{0,1\}^n \rightarrow \{0,1\}$ such that the output bit x_0 is

$$x_0 = \sum_{i=1}^{n} a_i x_i \text{, where } a_i \text{ is a coefficient } \epsilon \{0,1\}.$$

The feedback function has a period of 2^n-1. It is a common practice to employ primitive polynomials as a feedback function.

In [TCL09], the authors noted the following as the active bits in Trivium:

$$\{s_{66}, s_{69}, s_{93}\}\ \{s_{162}, s_{171}, s_{177}\}\ \{s_{243}, s_{264}, s_{288}\}.$$

Recognizing that each index is a multiple of 3, these bits can be generalized as

$$\{s_{3m_1}, s_{3m_2}, s_{3n_1}\}\ \{s_{3m_3}, s_{3m_4}, s_{3n_2}\}\ \{s_{3m_5}, s_{3m_6}, s_{3n_3}\}.$$

For the next part of the discussion, we are only concerned with the family of variables $\{m_1, m_2, n_1\}\ \{m_3, m_4, n_2\}\ \{m_5, m_6, n_3\}$.

If we consider these variables as powers of x for non-zero terms in a polynomial, we get the following:

$$x^{m_1} + x^{m_2} + x^{n_1} + x^{m_3} + x^{m_4} + x^{n_2} + x^{m_5} + x^{m_6} + x^{n_3}.$$

Definition (k-order primitive polynomial) Let $k \in \mathbb{N}$, a polynomial $p(x)$ is a k-order primitive polynomial if

$$p(x) = (x+1)^k q(x), \text{ where } q(x) \text{ is a primitive polynomial.}$$

[TCL09] described Trivium as a 3-order primitive polynomial with $q(x) = x^{22} + x^{23} + x^{31} + x^{54} + x^{57} + x^{59} + x^{81} + x^{88} + x^{96}$.

We were motivated by this definition to extend Trivium to a k^{th} round and use primitive polynomials to select the active state bits in Quadrivium. Certainly, our construction differentiates from the one proposed in [TCL09] in two major respects. First, Quadrivium is driven by the principles of PRNGs. This results in a pseudorandom sequence of lesser correlated bits. Second, the active state bits were redefined to be in agreement with concept of PRNGs.

Since Quadrivium takes a PRNG approach, our concern lies in the linearity of the pseudorandom output. We imposed several criteria to be in accordance with this approach. Recall in Trivium that the pseudorandom bit z_i is

$$s_{66} + s_{93} + s_{162} + s_{177} + s_{243} + s_{288}$$

In our construction, it was decided to redefine the active state bits to those responsible for pseudorandom bit z_i. Second, the active state bits must be derived from a primitive polynomial of degree 384. In Trivium, two state bits from each register is used to generate a single bit output. This should be the criterion to restrict the number of active state bits. As a result, the active state bits are s_{49}, s_{98}, s_{147}, s_{195}, s_{243}, s_{290}, s_{337} and s_{384}.

6.2 Algorithm

The initialization procedure, like Trivium, uses an 80-bit key and 80-bit IV. For the first and second registers, the key and IV is copied to the registers, respectively. The third register is filled

with zeroes excluding the last three state bits. Those are set to one. The last register is initialized with one-bit bit values except for the last four bits. They are zeroes.

$[s_1, s_2, \cdots, s_{98}] \leftarrow [K_1, \cdots, K_{80}, 0, \cdots, 0]$
$[s_{99}, s_{100}, \cdots, s_{195}] \leftarrow [IV_1, \cdots, IV_{80}, 0, \cdots, 0]$
$[s_{196}, s_{197}, \cdots, s_{290}] \leftarrow [0, \cdots, 0, 1, 1, 1]$
$[s_{291}, s_{292}, \cdots, s_{384}] \leftarrow [1, \cdots, 1, 0, 0, 0, 0]$

Once the registers are loaded, the **rotate** procedure is executed. The pseudocode is as follows:

FOR $i = 1$ to 1536
$t_1 \leftarrow s_{49} + s_{96} \cdot s_{97} + s_{98} + s_{171}$
$t_2 \leftarrow s_{147} + s_{193} \cdot s_{194} + s_{195} + s_{358}$
$t_3 \leftarrow s_{243} + s_{288} \cdot s_{289} + s_{290} + s_{69}$
$t_4 \leftarrow s_{337} + s_{382} \cdot s_{383} + s_{384} + s_{264}$

$[s_1, s_2 \cdots s_{98}] \leftarrow [t_2, s_1 \cdots s_{97}]$
$[s_{99}, s_{100} \cdots s_{195}] \leftarrow [t_4, s_{99} \cdots s_{194}]$
$[s_{196}, s_{197} \cdots s_{290}] \leftarrow [t_1, s_{196} \cdots s_{289}]$
$[s_{291}, s_{292} \cdots s_{384}] \leftarrow [t_3, s_{292} \cdots s_{383}]$

END FOR

The **rotate** procedure is reiterated for four full cycles.

The main procedure for Quadrivium is quite similar to Trivium. Below is the pseudocode:

//m = number of pseudorandom bits requested
FOR $i = 1$ to m
$t_1 \leftarrow s_{49} + s_{98}$
$t_2 \leftarrow s_{147} + s_{195}$
$t_3 \leftarrow s_{243} + s_{290}$
$t_4 \leftarrow s_{337} + s_{384}$
$z_i \leftarrow t_1 + t_2 + t_3 + t_4$

$t_1 \leftarrow s_{96} \cdot s_{97} + s_{171}$
$t_2 \leftarrow s_{193} \cdot s_{194} + s_{358}$
$t_3 \leftarrow s_{288} \cdot s_{289} + s_{69}$
$t_4 \leftarrow s_{382} \cdot s_{383} + s_{264}$
$[s_1, s_2 \cdots s_{98}] \leftarrow [t_2, s_1 \cdots s_{97}]$
$[s_{99}, s_{100} \cdots s_{195}] \leftarrow [t_4, s_{99} \cdots s_{194}]$
$[s_{196}, s_{197} \cdots s_{290}] \leftarrow [t_1, s_{196} \cdots s_{289}]$

$[s_{291}, s_{292} \cdots s_{384}] \leftarrow [t_3, s_{292} \cdots s_{383}]$

END FOR

Unlike Trivium, we did not include the previous values of t_i in the update function to produce the current values of t_i. The inclusion of the values does not necessarily have a negative impact on the generator. The decision to exclude these values was to restrict their influence to only the output bit versus both the output and the new state bits in Trivium. Both Quadrivium and Trivium has a nonlinear internal state so it is difficult to determine their periodicity. In [DP05], the authors noted that the period of Trivium is at least $2^{96-3} - 1$. This is under the assumption that the state evolves linearly. For Quadrivium, the period is at least $2^{384} - 1$, given the same assumption. This is based on the fact that the output function is derived from a primitive polynomial.

Chapter 7
Statistical Comparisons

Statistical testing is one of the most common methods used to determine the output quality of PRNGs. This analysis employed NIST statistical test suite (STS) and Dieharder: A Random Number Test Suite[11] to assess the performance of OpenJDK SecureRandom, Trivium and Quadrivium. We used SecureRandom and Trivium as a benchmark for the performance of Quadrivium. For all analyses, three different pseudorandom data files from each generator were tested. Each file consisted of 122.88 million bytes. This was determined by the Dieharder test suite which requires about 31 million integers for proper analysis.

7.1 NIST Statistical Test Suite

National Institute of Standards and Technology (NIST) statistical test suite for cryptographically secure RNGs and PRNGs is a standard for statistical testing. The suite contains fifteen tests[12] which analyze the quality of a PRNG's output; and determine whether the outputs mimic the behaviors of truly random sequences. Each test uses a test statistic to determine whether to reject the null hypothesis or not. The null hypothesis is the tested sequence is random; it lacks a pattern and portrays irregularity. The alternative hypothesis is the sequence is not random, a pattern was detected therefore it is predictable.

The assessments focus on different behaviors which indicate predictability in a sequence; they can be classified into four main types. The first type is frequency tests. They are the Frequency test (FREQ), Frequency Test within a Block (BLOCKFREQ), Runs Test (RUNS), Test for the Longest Run of Ones in a Block (LONGRUNS). The following two tests, Binary Matrix Rank Test (RANK) and Discrete Fourier Transform Test (FFT) fall under the repetitive patterns type. Non-overlapping Template Matching Test, Overlapping Template Matching Test (OTEMP), Maurer's "Universal Statistical" Test (UNIVERSAL), Linear Complexity Test (LINCOMP), Serial Test (SERIAL) and Approximate Entropy Test (APPROXENT) are pattern matching types. The fourth type is random walks and consists of Cumulative Sums Test (CUSUMS), Random Excursions Test and Random Excursions Variant Test. [ZG12]

Even though all tests focus on different aspects of an ensemble, there are three assumptions that they all hold about random outputs. These assumptions are taken in consideration when determining the quality of a PRNG's outputs and if they are comparable to a set of truly random sequences. The assumptions are uniformity, scalability and consistency. Looking at a random sequence of length n, uniformity means the occurrence of zeroes should be one-half of the sequence, likewise the occurrence of ones. Scalability determines to what degree is a sequence random. This property also expects that all subsets of a random sequence must also be random. Consistency expresses the behavior of a PRNG. According to the literature, a consistent PRNG will always produce the random sequences of equal quality.

---

[11] This suite includes tests from Diehard Battery of Tests. To refrain from redundancy, we conducted analyses from both suites with only Dieharder.

[12] The Document included sixteen tests but the actual program performs only fifteen. This paper will discuss all sixteen tests.

It is not necessary to conduct all tests in the suite when analyzing a PRNG. The analyst is responsible for selecting the appropriate combination of assessments used to study a generator.

7.2 STS Results

Twelve tests in the suite were conducted on each file. We excluded the Non-overlapping Template Matching Test, Random Excursions Test and Random Excursions Variant Test. For testing purposes the data file was segmented into 980 subsequences, each one million bits in length. The significance level, $\alpha = 0.01$, was the determining factor for the number of subsequences. For a significance level of 0.01, there should be at least one hundred sequences available for testing. The subsequence length was chosen based on the Linear Complexity test. This test requires, at minimum, one million bit-long sequences to function properly. This is the largest quantity amongst all the tests in the suite.

For each STS run, the suite returns twelve values for each test. One value is the proportion of subsequences passing the respective test. Another value is a single p-value of all the p-values determined. The remaining values are the distribution of p-values over ten subintervals on (0, 1]. The p-value is used to determine the degree of uniformity amongst sequences. The closer a p-value is to one; the closer it is to perfect uniformity. A p-value ≥ 0.0001 and a proportions value of 0.979592 are required to pass a test.

For Figures 3 to 6, we selected the best performing dataset for each generator and compared the distribution of p-values for four tests. The best performing dataset for each generator happened to be dataset 3. The p-values of the sequences are distributed over ten bins. Bin 1 reflects p-values ranging from zero up to but not including 0.1. Bin 2 sequences ranges from 0.1 up to but not including 0.2. The range of each successive bin is incremented by a tenth.

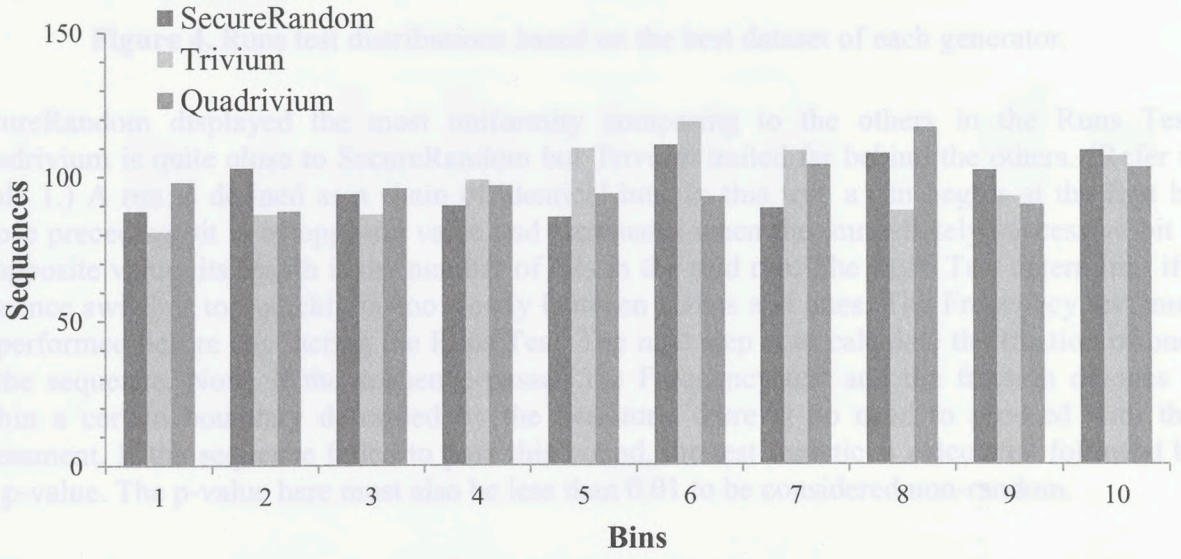

Figure 3. Frequency test distributions based on the best dataset of each generator.

Quadrivium and SecureRandom degree of uniformity are approximately the same for the Frequency test. They are significantly larger than Trivium degree of uniformity. (Refer to Table

1.) The Frequency (Monobit) Test verifies that the existence of zeroes and ones—bit-wise— are equally proportional. In relation to the other tests in the suite, this is by far the most important; if an ensemble cannot pass this test, most likely it will not pass the other fifteen. Remember, one of the assumptions of truly random sequence is uniform distribution of zeroes and ones and another is scalability. Thus the number of zeroes and ones must be balanced bit-wise in order to be balanced word-wise. The test takes a sequence and converts all the zero bits to a value of negative one (-1) and all the one bits to positive one (+1). The values are all added together. This value will be used to compute the test statistic. After the test statistic is determined, it is used to calculate the p-value. If the p-value is greater than or equal to 0.01 the sequence is said to be random.

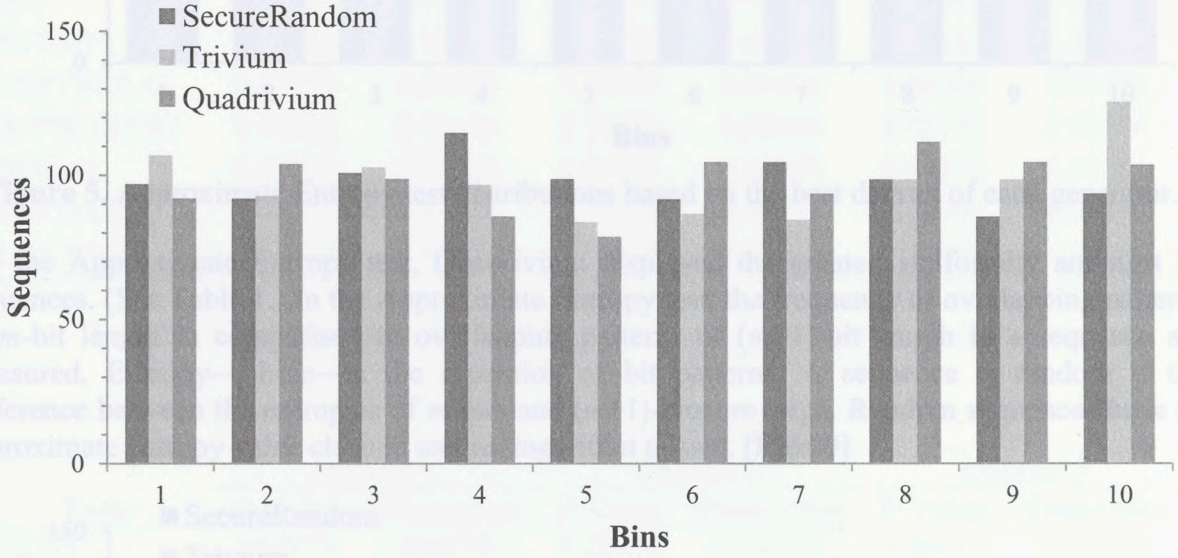

Figure 4. Runs test distributions based on the best dataset of each generator.

SecureRandom displayed the most uniformity comparing to the others in the Runs Test. Quadrivium is quite close to SecureRandom but Trivium trailed far behind the others. (Refer to Table 1.) A run is defined as a chain of identical bits. In this test, a run begins at the first bit whose preceding bit is of opposite value and terminates when the immediately successive bit is of opposite value; its length is the number of bits in the said run. The Runs Test determines if a sequence switches too quickly or too slowly between zeroes and ones. The Frequency test must be performed before conducting the Runs Test. The next step is to calculate the fraction of ones in the sequence. Now, if the sequence passed the Frequency test and the fraction of ones is within a certain boundary described by the literature, there is no need to proceed with this assessment. If the sequence failed to pass this bound, the test statistic is calculated followed by the p-value. The p-value here must also be less than 0.01 to be considered non-random.

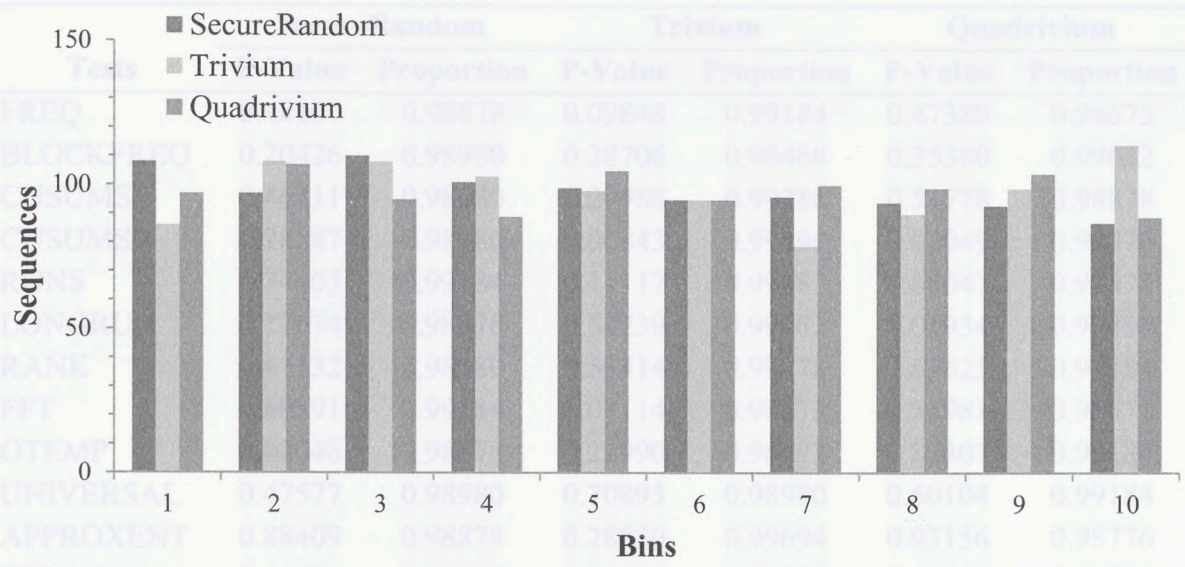

Figure 5. Approximate Entropy test distributions based on the best dataset of each generator.

For the Approximate Entropy test, Quadrivium displayed the greatest uniformity amongst its sequences. (See Table 1.) In the Approximate Entropy test, the frequency of overlapping patterns of m-bit length in comparison to overlapping patterns of $(m+1)$-bit length in a sequence are measured. Entropy— here—is the repetition of bit patterns. A sequence is random if the difference between the entropies of m-bits and $(m+1)$-bits are large. Random sequences have an approximate entropy value close to natural logarithm of two. [Ruk00]

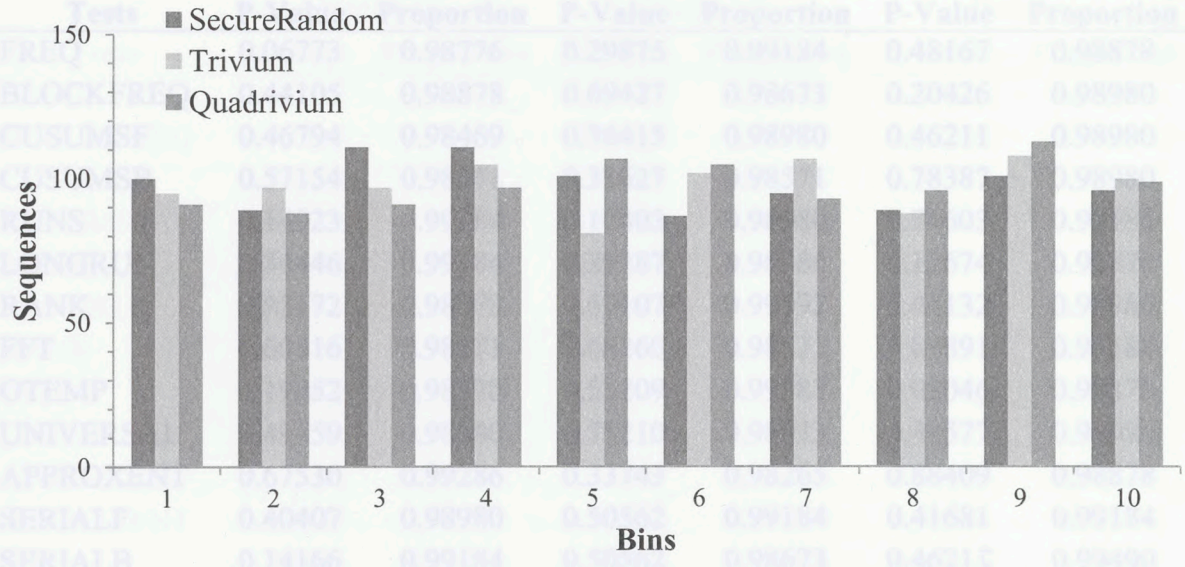

Figure 6. Linear Complexity test distributions based on the best dataset of each generator.

Quadrivium is the closest generator to a uniform distribution. (Refer to Table 1.) The Linear Complexity test performs the Berlekamp-Massey Algorithm on each block to find the shortest LFSR within a block. Short LFSRs indicate that the sequence is not random.

Tests	SecureRandom		Trivium		Quadrivium	
	P-Value	Proportion	P-Value	Proportion	P-Value	Proportion
FREQ	0.48167	0.98878	0.09848	0.99184	0.47380	0.98673
BLOCKFREQ	0.20426	0.98980	0.28706	0.98469	0.35380	0.99082
CUSUMSF	0.46211	0.98980	0.20988	0.99286	0.51778	0.98878
CUSUMSB	0.78387	0.98980	0.00843	0.99490	0.01049	0.98776
RUNS	0.74603	0.99184	0.11117	0.99082	0.66047	0.98571
LONGRUN	0.12674	0.98878	0.54239	0.99082	0.01934	0.99490
RANK	0.41132	0.98980	0.53414	0.98878	0.65623	0.99184
FFT	0.89891	0.99184	0.03114	0.98673	0.51981	0.98571
OTEMP	0.02046	0.98878	0.22990	0.98673	0.80307	0.99184
UNIVERSAL	0.47577	0.98980	0.70893	0.98980	0.60104	0.99184
APPROXENT	0.88409	0.98878	0.28850	0.99694	0.93156	0.98776
SERIAL	0.41681	0.99184	0.93038	0.98980	0.67319	0.99184
SERIALB	0.46211	0.99490	0.77604	0.99082	0.28706	0.99184
LINCOMP	0.66895	0.98673	0.67530	0.98367	0.77801	0.99286

Table 1. STS tests results from the best performing dataset of each generator.

Tests	SecureRandom					
	Dataset 1		Dataset 2		Dataset 3	
	P-Value	Proportion	P-Value	Proportion	P-Value	Proportion
FREQ	0.06773	0.98776	0.29875	0.99184	0.48167	0.98878
BLOCKFREQ	0.44105	0.98878	0.69427	0.98673	0.20426	0.98980
CUSUMSF	0.46794	0.98469	0.76415	0.98980	0.46211	0.98980
CUSUMSB	0.57154	0.98571	0.32627	0.98571	0.78387	0.98980
RUNS	0.14923	0.99184	0.10403	0.98980	0.74603	0.99184
LONGRUN	0.54446	0.99184	0.29287	0.98980	0.12674	0.98878
RANK	0.85172	0.98878	0.67107	0.99592	0.41132	0.98980
FFT	0.60316	0.98571	0.66260	0.98571	0.89891	0.99184
OTEMP	0.19552	0.98878	0.53209	0.99388	0.02046	0.98878
UNIVERSAL	0.49759	0.98980	0.75210	0.98673	0.47577	0.98980
APPROXENT	0.67530	0.99286	0.33743	0.98265	0.88409	0.98878
SERIALF	0.40407	0.98980	0.50562	0.99184	0.41681	0.99184
SERIALB	0.14166	0.99184	0.50562	0.98673	0.46211	0.99490
LINCOMP	0.45824	0.98878	0.72554	0.98878	0.66895	0.98673

Table 2. STS tests results of all SecureRandom datasets.

	Trivium					
	Dataset 1		Dataset 2		Dataset 3	
Tests	P-Value	Proportion	P-Value	Proportion	P-Value	Proportion
FREQ	0.49959	0.99184	0.23606	0.98673	0.09848	0.99184
BLOCKFREQ	0.24235	0.98571	0.56317	0.98878	0.28706	0.98469
CUSUMSF	0.46989	0.98776	0.19552	0.98673	0.20988	0.99286
CUSUMSB	0.61800	0.99184	0.01962	0.98673	0.00843	0.99490
RUNS	0.46599	0.99184	0.63924	0.98673	0.11117	0.99082
LONGRUN	0.15624	0.99184	0.53003	0.98878	0.54239	0.99082
RANK	0.29433	0.98878	0.42977	0.98878	0.53414	0.98878
FFT	0.13840	0.98878	0.73582	0.98571	0.03114	0.98673
OTEMP	0.83961	0.98469	0.00118	0.99490	0.22990	0.98673
UNIVERSAL	0.37067	0.98571	0.98375	0.99082	0.70893	0.98980
APPROXENT	0.13521	0.99592	0.14752	0.97959	0.28850	0.99694
SERIALF	0.16539	0.98878	0.18708	0.99286	0.93038	0.98980
SERIALB	0.26740	0.98776	0.81801	0.99388	0.77604	0.99082
LINCOMP	0.47185	0.99286	0.02698	0.99082	0.67530	0.98367

Table 3. STS tests results of all Trivium datasets.

	Quadrivium					
	Dataset 1		Dataset 2		Dataset 3	
Tests	P-Value	Proportion	P-Value	Proportion	P-Value	Proportion
FREQ	0.07496	0.98673	0.48563	0.99184	0.47380	0.98673
BLOCKFREQ	0.93729	0.98980	0.02625	0.98776	0.35380	0.99082
CUSUMSF	0.24362	0.98878	0.79546	0.98878	0.51778	0.98878
CUSUMSB	0.43539	0.98367	0.85342	0.99082	0.01049	0.98776
RUNS	0.01194	0.98673	0.36896	0.98980	0.66047	0.98571
LONGRUN	0.21101	0.99286	0.22267	0.99184	0.01934	0.99490
RANK	0.91141	0.99082	0.71310	0.98878	0.65623	0.99184
FFT	0.17498	0.98163	0.04048	0.98980	0.51981	0.98571
OTEMP	0.46018	0.98673	0.76615	0.99490	0.80307	0.99184
UNIVERSAL	0.51169	0.98163	0.84311	0.99388	0.60104	0.99184
APPROXENT	0.89309	0.99286	0.35880	0.98367	0.93156	0.98776
SERIALF	0.55900	0.98776	0.94486	0.99082	0.67319	0.99184
SERIALB	0.98424	0.98061	0.30173	0.99286	0.28706	0.99184
LINCOMP	0.22990	0.99184	0.65835	0.98776	0.77801	0.99286

Table 4. STS tests results of all Quadrivium datasets.

Tests	Average proportions		
	SecureRandom	Trivium	Quadrivium
FREQ	0.98946	0.99014	0.98844
BLOCKFREQ	0.98844	0.98639	0.98946
CUSUMSF	0.98810	0.98912	0.98878
CUSUMSB	0.98707	0.99116	0.98741
RUNS	0.99116	0.98980	0.98741
LONGRUN	0.99014	0.99048	0.99320
RANK	0.99150	0.98878	0.99048
FFT	0.98776	0.98707	0.98571
OTEMP	0.99048	0.98878	0.99116
UNIVERSAL	0.98878	0.98878	0.98912
APPROXENT	0.98810	0.99082	0.98810
SERIALF	0.99116	0.99048	0.99014
SERIALB	0.99116	0.99082	0.98844
LINCOMP	0.98810	0.98912	0.99082

Table 5. The average proportions of all generators for each STS test.

Table 1 shows a comparison of the best performing dataset from each generator. The datasets were chosen based on the dataset with the highest overall average of proportions for each generator. Tables 2 through 4 displays the overall p-value of p-values and proportion for each test from all datasets. Table 5 shows the average proportions for each generator. SecureRandom had the highest average for the runs, Binary Matrix Rank, Discrete Fourier Transform and Serial Tests. Trivium outperformed the others on the Frequency, Cumulative Sums and Approximate Entropy Tests. Quadrivium had the highest average proportions for the Frequency within a Block, Tests for the Longest Runs, Overlapping Template Matching, Maurer's "Universal Statistical" Test and Linear Complexity Tests. (Refer to Appendix C for test descriptions.)

7.3 Diehard Battery of Test

Diehard is a statistical testing suite created by George Marsaglia, who is also the creator of pseudorandom number generator Xorshift. Diehard includes sixteen tests – fifteen personally authored by Marsaglia– that gauge the randomness quality of a generator. The tests require a binary file of at least 80 million random bits as input. Each test in Diehard has its own set of parameters. In other words, the number of bits needed for testing varies. Additionally, there is an improved testing suite available known as Dieharder battery of tests.

A majority of the assessments in the suite uses a p-value to determine if a sequence is random. This is similar to the NIST statistical test suite which also has a number of tests that rely on p-values to draw a conclusion. In statistics, p-values represent the probabilities that some arbitrary event will occur; their purpose is to accept or reject the null hypothesis, which is the tested claim. In Diehard, the null hypothesis is the analyzed sequence is random. Tested sequences are

acknowledged as random if p-values are not close to zero or one. Contrarily, in NIST statistical test suite, the further the p-value is to one, the further a sequence is to being truly random.

Another difference between the two testing suites is the analysis of the results. NIST statistical test suite specifies the p-value needed to reject the null hypothesis. Diehard battery of tests is ambiguous and only states that the p-values should be uniform on the set [0, 1).

The names of the exams included in the Diehard battery of tests are Birthday Spacings Test; Overlapping 5-Permutation Test; separate Binary Rank Tests for 31x31, 32x32 and 6x8 matrices; Bitstream Test; OPSO, OQSO and DNA (Overlapping Pairs Sparse Occupancy, Overlapping Quadruples Sparse Occupancy and DNA Test, respectively); separate Count the 1s Test for byte-streams and specific bytes; This is a Parking Lot Test; Minimum Distance Test; 3DSpheres Test; Squeeze Test; Overlapping Sums Test; Runs test –which is a standard test; and Craps Test.

7.4 Dieharder: A Random Number Test Suite

Dieharder is a test pack for random number generators. The suite includes modified tests from Diehard battery of tests, NIST statistical test suite as well as some assessments created by Robert G. Brown, the chief developer of Dieharder. The test suite is an open source project whose purpose is to be a one-stop source for quantifying randomness. The project encourages inclusion of other new testing schemes from other developers. Dieharder is primarily concerned with analyzing the randomness quality and speed of truly random and pseudorandom number generators.

In comparison to STS and Diehard, the suite prefers to examine the actual generator, not a random output file produced by the generator. The reasoning behind this is "perfect randomness is the production of 'unlikely' sequences of random number at an average rate." Looking at the output alone is not sufficient to declare randomness; the likelihood of the sequence as a whole cannot be determined. Even though Dieharder prefers the aforementioned method of testing, it can still accommodate file-based inputs.

Dieharder includes tests from both STS and Diehard. Parameters from both suites are altered so failures are concluded without ambiguity. Moreover, the Diehard tests are improved in three ways. One, assessments that uses KSTEST, Kolmogorov-Smirnov test, imposes a higher default quantity of one hundred p-values. This coincides with Dieharder's aim to determine unambiguous failure. Two, analysts have more control over tests that use samples. Sample sizes are treated as a variable rather than a fixed constant. Three, assessments that employs overlapping techniques on sequences were adjusted to use non-overlapping techniques. Please note that these improvements were made only if it was possible.

There are ten tests included in Dieharder that were created by Robert G. Brown. They are the following: Bit Distribution Test, Generalized Minimum Distance Test, Permutations Test, Lagged Sums Test, KSTest (Kolmogorov-Smirnov Test) Test, DAB Byte Distribution Test, DCT (Frequency Analysis), DAB Fill Tree Test, DAB Fill Tree 2 Test and DAB Monobit Test.

7.5 Dieharder Results

Diehard Battery of tests and RGB tests were conducted under the Dieharder test suite. Each dataset was parsed into 32 bit unsigned integers totaling 30.72 million integers. The suite returned two results: a p-value and an assessment of passed, weak or failed. A weak assessment signifies that the p-value ≤ 0.005. A failed assessment signifies the p-value ≤ 0.000001.

Tables 6 and 7 shows the assessment counts for all collected data. Even though Diehard and RGB are sets of fifteen and ten tests, respectively, the total assessment counts are greater. This is attributed to the fact that some of the tests are conducted with multiple parameters. One of the RGB tests, Lagged Sums Test, for example, has 33 different variants.

	Diehard tests assessment count								
	SecureRandom			Trivium			Quadrivium		
Dataset	1	2	3	1	2	3	1	2	3
Assessment									
PASSED	19	17	16	16	19	17	18	16	18
WEAK	0	2	3	2	0	2	1	3	1
FAILED	0	0	0	1	0	0	0	0	0

Table 6. Summary of all Diehard tests assessments.

All generators had one data set that was considered weak for the OPSO test, SecureRandom dataset 3, Trivium dataset 1 and Quadrivium dataset 2. The Binary Matrix Rank Test 32x32 was also a common problem for all generators. Trivium failed this test with dataset 1 and was considered weak for dataset 3. Quadrivium received a weak assessment for data sets 1 and 2 while SecureRandom received a weak assessment for data set 3. The other weak assessments are as follows: SecureRandom dataset 2, Count the ones test for byte-streams and bytes; SecureRandom dataset 3, Count the ones test for bytes; Trivium dataset 1, Craps 2 test; Trivium dataset 3, Count the ones test for bytes; Quadrivium dataset 2, Runs test; Quadrivium dataset 3, OQSO. (Refer to Table 6 and Appendix C for test descriptions.)

	RGB tests assessment count								
	SecureRandom			Trivium			Quadrivium		
Dataset	1	2	3	1	2	3	1	2	3
Assessment									
PASSED	47	46	46	36	43	30	46	42	42
WEAK	7	4	4	16	6	18	6	9	4
FAILED	7	11	11	9	12	13	9	10	15

Table 7. Summary of RGB tests assessments.

There were four RGB tests that all generators failed consistently. They are the DAB Byte Distribution, DAB Monobit 2 and, Lagged Sums 29 and 31 tests. Individually, SecureRandom consistently failed the Lagged Sums 23 test; Trivium always failed Lagged Sums 9 and 23 tests; and as for Quadrivium, the Lagged Sums 9, 14, 19 and 24 tests always resulted in failure. All other failures are some variant of the Lagged Sums Test. For Trivium and Quadrivium, the weak assessments were mainly Lagged Sums Test variants and a couple of Bit Distribution tests. SecureRandom weak assessments were only Lagged Sums Tests. Trivium was the only generator to have a consistent weak assessment for a particular test. It was the Lagged Sums 15 test. (Refer to Table 7 and Appendix C for test descriptions.)

	Generation time in milliseconds		
	SecureRandom	**Trivium**	**Quadrivium**
Dataset 1	2442	1775836	2394939
Dataset 2	2652	1807494	2598758
Dataset 3	2653	1777542	2409478

Table 8. Generation times for 122.88 million bytes.

In Table 8, we see that the generation times of Trivium and Quadrivium are significantly slower than SecureRandom. This can be attributed to the fact the implementations of Trivium and Quadrivium are non-optimized. The codes, see Appendix A and B, are based on direct translations of the pseudocode presented in [DP05]. SecureRandom is an official implementation available in the Java Library therefore its code has been created with efficiency.

Conclusion

In this paper, we presented a revised model of Trivium that focused on improving the selection of state bits. We considered the entire state of Trivium to make improvements versus its individual registers in previous works. We were aware that the unpredictability of pseudorandom sequences is directly correlated to the entire set of state bits selected to yield a stream bits and proposed a solution in our model.

The analyses we presented indicates that Quadrivium statistically outperforms Trivium exhibiting more characteristics of uniformity. Referring to Tables 3 and 4, we see that the Quadrivium had more p-values closer to one overall and amongst individual tests. Quadrivium also had a higher passing rate on the Diehard and RGB tests. (See Tables 6 and 7.) Additionally, Quadrivium performs relatively as well as SecureRandom which employs a complex generating algorithm. There was not a significant difference in the average number of sequences passing the STS; or the average number of passed, weak and failed assessments for the Diehard and RGB tests. We can also conclude that Quadrivium consistently performs well on tests that checks for linear complexity, pattern matching and pseudorandomness on a sequence-level.

Even though Quadrivium shows potential, there are some lacking points about the generator. First, Quadrivium did not do substantially well on tests that check for correlation on a bit level. We also see in Table 8 that the generation time for Quadrivium is the slowest comparing to the other generators. Future work can be to find solutions to lessen bit correlations as well as implementing optimized algorithms in the code. Potential research could also be to determine the period and security of Quadrivium.

Bibliography

[Barak] Barak B. "Lecture 4 - Computational Indistinguishability ..." Lecture 4 - Computational Indistinguishability, Pseudorandom Generators. Princeton University, n.d. Web. 12 Nov. 2016.

[Bog+07] Bogdanov A, et al. "PRESENT: An ultra-lightweight block cipher." International Workshop on Cryptographic Hardware and Embedded Systems. Springer Berlin Heidelberg, 2007.

[Boon] Boon M. "Lecture 5: Random-number Generators." 2WB05 Simulation. Eindhoven University of Technology. 6 Dec. 2012. Simulation Lecture 5. Web.

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, Advances in Cryptology – EUROCRYPT 2006, volume 4004 of Lecture Notes in Computer Science, pages 409–426, St. Petersburg, Russia, May 28 – June 1, 2006. Springer, Berlin, Germany.

[Debian] Debian.org. (2008). Debian -- Security Information -- DSA-1571-1 openssl. [online] Available at: http://www.debian.org/security/2008/dsa-1571 [Accessed 1 Oct. 2015].

[dHG08] de Koning Gans G, Hoepman J, and Garcia F. "A Practical Attack on the MIFARE Classic." Smart Card Research and Advanced Applications. Springer Berlin Heidelberg, 2008. 267-282.

[Dodis] Dodis Y. "Randomness and Cryptography." Microsoft PowerPoint - Random1. Web.

[DP05] DeCannière C and Preneel B. "Trivium specifications. eSTREAM." ECRYPT Stream Cipher Project, Report 30.2005 (2005): 266.

[DeC06] De Cannière C. "Trivium: A stream cipher construction inspired by block cipher design principles." *International Conference on Information Security*. Springer Berlin Heidelberg, 2006.

[DPR+13] Dodis Y, Pointcheval D, Ruhault S, Vergnaud D and Wichs D, "Security analysis of pseudo-random generators with input: /dev/random is not robust", 2013.

[Dwo01] Dworkin, M. Recommendation for block cipher modes of operation. methods and techniques. No. NIST-SP-800-38A. National Institute of Standards and Technology Gaithersburg MD Computer Security Div, 2001.

[ECS05] Eastlake D, Crocker S and Schiller J. "RFC 4086 Randomness Requirements for Security, June 2005." *tools. ietf. org/html/rfc4086.*

[FIPS01] PUB, FIPS. *Security Requirements for Cryptographic Modules*. Diss. National Institute of Standards and Technology, 2001.

[FN03] Ferguson N and Schneier B. *Practical cryptography*. Vol. 23. New York: Wiley, 2003.

[Gol10] Goldreich O. "A primer on pseudorandom generators." Vol. 55. American Mathematical Society, 2010.

[Gol90] Goldreich O. "A note on computational indistinguishability." Information Processing Letters 34.6 (1990): 277-281.

[GPR06] Gutterman Z, Pinkas B and Reinman T. "Analysis of the Linux random number generator." In 2006 IEEE Symposium on Security and Privacy, pages 371–385, Berkeley, California, USA, May 21–24, 2006. IEEE Computer Society Press.

[GW96] Goldberg I and Wagner D. "Randomness and the Netscape browser."Dr Dobb's Journal-Software Tools for the Professional Programmer 21.1 (1996): 66-71.

[Jag08] Jagannatam A. "Mersenne Twister–A Pseudo Random Number Generator and its variants." *George Mason University, Department of Electrical and Computer Engineering* (2008).

[KSW+98] Kelsey J, Schneier B., Wagner D and Hall C. "Cryptanalytic attacks on pseudorandom number generators". FSE. Lecture Notes in Computer Science, Springer (1998)

[Mar03] Marsaglia, George. "Xorshift rngs." *Journal of Statistical Software* 8.14 (2003): 1-6.

[MBT+17] McKay K, Bassham L, Turan and Mouha N. "Report on lightweight cryptography." NISTIR 8114 (2017).

[MCC+06] McEvoy, Robert, et al. "Fortuna: cryptographically secure pseudo-random number generation in software and hardware." *Irish Signals and Systems Conference, 2006. IET.* IET, 2006.

[MGH13] Melia-Segui J, Garcia-Alfaro J and Herrera-Joancomarti J. "J3Gen: A PRNG for low-cost passive RFID." Sensors 13.3 (2013): 3816-3830.

[MK92] Matsumoto M and Kurita Y. "Twisted GFSR generators."*ACM Transactions on Modeling and Computer Simulation (TOMACS)* 2.3 (1992): 179-194.

[MMS13] Michaelis K, Meyer C and Schwenk J. "Randomly failed! The state of randomness in current Java implementations." Topics in Cryptology–CT-RSA 2013. Springer Berlin Heidelberg, 2013. 129-144.

[OracleRa] "Random (Java Platform SE 8)." *Random (Java Platform SE 8)*. Web. N.p., n.d. 01 May 2016.

[OracleSR] "SecureRandom (Java Platform SE 8)." *SecureRandom (Java Platform SE 8)*. N.p., n.d. Web. 15 May 2016.

[PP10] Paar. Christof and Jan Pelzl. "Stream Ciphers." *Understanding Cryptography*. S.l.: Springer, 2010. 29-54. Print.

[PS96] Pincus S and Singer B. "Randomness and degrees of irregularity." *Proceedings of the National Academy of Sciences* 93.5 (1996): 2083-2088.

[Roe05] Roeck A. "Pseudorandom number generators for cryptographic applications". 2005.

[Ruk00] Rukhin, Andrew L. "Approximate entropy for testing randomness." *Journal of Applied Probability* 37.1 (2000): 88-100.

[Ruk+10] Rukhin A, et al. "A statistical test suite for random and pseudorandom number generators for cryptographic applications Revision 1a. NIST Special Publication 800-22". 2010.

[Sha48] Shannon C. "A mathematical theory of communication." ACM SIGMOBILE Mobile Computing and Communications Review 5.1 (2001): 3-55.

[Sta11] Stamp M. "Random Numbers." Information Security: Principles and Practice. 2nd ed. Hoboken, NJ: Wiley-Interscience, 2011. 145-46. Print.

[TCL09] Tian Y, Chen G and Li J. "On the design of Trivium." IACR Cryptology ePrint Archive 2009 (2009): 431.

[Vad12] Vadhan S, "Pseudorandomness." Foundations and Trends in Theoretical Computer Science: Vol. 7: No. 1–3, pp 1-336, 2012.

[Von51] Von Neumann J. "13. Various Techniques Used in Connection With Random Digits." (1951).

[Xiannong] Xiannong M. "Linear Congruential Method." Linear Congruential Method. Bucknell University, 18 Oct. 2002. Web. 10 Sept. 2016.

[ZG12] Zaman, J. K. M., and Ranjan Ghosh. "A review study of NIST Statistical Test Suite: Development of an indigenous computer package." *arXiv preprint arXiv:1208.5740* (2012).

Appendix A

Quadrivium.java

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.BitSet;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;

/**
 * This class provides a pseudorandom number generator.
 * @author Latoya
 */
public class Quadrivium {

    private BitSet state = new BitSet(384);

    public Quadrivium() {
        init();
    }

    public Quadrivium(byte[] key, byte[] IV) {
        init(key, IV);
    }

    /* Get key and IV, initialize generator.*/

    private void init() {
        final byte[] key = new byte[10];
        final byte[] IV = new byte[10];
        Key.generateKey(key);
        InitVector.getIV(IV);
        init(key, IV);
    }//end init method

    /*Copies key and IV to state, then initialize generator.*/

    private void init(byte[] key, byte[] IV) {
        byte[] bytebuffer = new byte[48];
        int currIndex = 0;
        int index = 0;

        //load Register A
        for (int i = 0; i < key.length; i++) {

            bytebuffer[currIndex++] = key[i];
        }
```

```
        bytebuffer[currIndex++] = 0;
        bytebuffer[currIndex++] = 0;

        //copy the first two bits of the IV
        byte joint = IV[index++];
        bytebuffer[currIndex++] = (byte) ((joint & 0xff) >>> 2);

        //copy the rest of IV to register B
        while (index < IV.length) {
            byte tmp = IV[index - 1];
            byte tmp2 = IV[index];

            tmp = (byte) (tmp << 6);
            tmp2 = (byte) ((tmp & 0xff) >>> 2);
            //bit addition
            int copy = tmp + tmp2;
            //new byte
            bytebuffer[currIndex++] = (byte) copy;
            index++;
        }

        joint = IV[IV.length - 1];
        bytebuffer[currIndex++] = (byte) (joint << 6);

        //Register C
        while (currIndex < 35) {
            bytebuffer[currIndex++] = 0;
        }
         bytebuffer[currIndex++] = 1;

        //Register D
        while (currIndex < bytebuffer.length - 1) {
            bytebuffer[currIndex++] = -1;
        }
        bytebuffer[currIndex] = -16;
        state = BitSet.valueOf(bytebuffer);
        rotate();

    }//end init method (with key and IV input)

    /* Generates specified number of bytes and update state.*/

    public synchronized void getBytes(byte[] output) {
        boolean t1, t2, t3, t4;
        boolean[] z = new boolean[8];
        int i = 0, index = 0, count = 0, b = output.length * 8;

        boolean s49, s69, s96, s97, s98,
                s147, s171, s193, s194, s195,
                s243, s264, s288, s289, s290,
```

```
        s337, s358, s382, s383, s384;

    while (count < b) {
        s49 = state.get(48);
        s69 = state.get(68);
        s96 = state.get(95);
        s97 = state.get(96);
        s98 = state.get(97);
        s147 = state.get(146);
        s171 = state.get(170);
        s193 = state.get(192);
        s194 = state.get(193);
        s195 = state.get(194);
        s243 = state.get(242);
        s264 = state.get(263);
        s288 = state.get(287);
        s289 = state.get(288);
        s290 = state.get(289);
        s337 = state.get(336);
        s358 = state.get(358);
        s382 = state.get(381);
        s383 = state.get(382);
        s384 = state.get(383);

        //generate bit
        t1 = s49 ^ s98;
        t2 = s147 ^ s195;
        t3 = s243 ^ s290;
        t4 = s337 ^ s384;
        z[i] = t1 ^ t2 ^ t3 ^ t4;

        //update
        t1 = s96 & s97 ^ s171;
        t2 = s193 & s194 ^ s358;
        t3 = s288 & s289 ^ s69;
        t4 = s382 & s383 ^ s264;

        //shift register A
        for (int j = 96; j >= 0; j--) {
            state.set(j + 1, state.get(j));
        }
        state.set(0, t2);
        //shift register B
        for (int j = 193; j >= 98; j--) {
            state.set(j + 1, state.get(j));
        }
        state.set(98, t4);
        //shift register C
        for (int j = 288; j >= 195; j--) {
            state.set(j + 1, state.get(j));
```

```
            }
            state.set(195, t1);

            //shift register D
            for (int j = 382; j >= 290; j--) {
                state.set(j + 1, state.get(j));
            }
            state.set(290, t3);
            i++;

            //copy generated byte to state
            if (i == 8) {
                for (int j = 0; j < 8; j++) {
                    if (z[j]) {
                        output[index] |= (byte) (1 << (7 - j));
                        z[j] = false;//reset
                    }
                }
                index++;
                i = 0; //reset z index
            }
            count++;
        }
    }//end getBytes method

    /* Rotates the state bits over four cycles. */

    private void rotate() {
        boolean t1, t2, t3, t4;
        int i = 0;
        int cycles = (4 * state.length());

        boolean s49, s69, s96, s97, s98,
                s147, s171, s193, s194, s195,
                s243, s264, s288, s289, s290,
                s337, s358, s382, s383, s384;

        while (i <= cycles) {
            s49 = state.get(48);
            s69 = state.get(68);
            s96 = state.get(95);
            s97 = state.get(96);
            s98 = state.get(97);
            s147 = state.get(146);
            s171 = state.get(170);
            s193 = state.get(192);
            s194 = state.get(193);
            s195 = state.get(194);
            s243 = state.get(242);
            s264 = state.get(263);
```

```
            s288 = state.get(287);
            s289 = state.get(288);
            s290 = state.get(289);
            s337 = state.get(336);
            s358 = state.get(358);
            s382 = state.get(381);
            s383 = state.get(382);
            s384 = state.get(383);

            t1 = s49 ^ s96 & s97 ^ s98 ^ s171;
            t2 = s147 ^ s193 & s194 ^ s195 ^ s358;
            t3 = s243 ^ s288 & s289 ^ s290 ^ s69;
            t4 = s337 ^ s382 & s383 ^ s384 ^ s264;

            //shift register A
            for (int j = 96; j >= 0; j--) {
                state.set(j + 1, state.get(j));
            }
            state.set(0, t2); //s1
            //shift register B
            for (int j = 193; j >= 98; j--) {
                state.set(j + 1, state.get(j));
            }
            state.set(98, t4);//s99
            //shift register C
            for (int j = 288; j >= 195; j--) {
                state.set(j + 1, state.get(j));
            }
            state.set(195, t1);//s196
            //shift register D
            for (int j = 382; j >= 290; j--) {
                state.set(j + 1, state.get(j));
            }
            state.set(290, t3);//s291
            i++;
        }

    }//end rotate method

}//end Quadrivium Class
```

Appendix B

Trivium.java

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.BitSet;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;

/**
 * This class provides a pseudorandom number generator.
 * @author Latoya
 */
public class Trivium {

    private BitSet state = new BitSet(288);

    public Trivium() {
        init();
    }

    public Trivium(byte[] key, byte[] IV) {
        init(key, IV);
    }

    /* Get key and IV, initialize generator.*/
    private void init() {
        final byte[] key = new byte[10];
        final byte[] IV = new byte[10];
        Key.generateKey(key);
        InitVector.getIV(IV);
        init(key, IV);
    }//end init method

    /*Copies key and IV to state, then initialize generator.*/

    private void init(byte[] key, byte[] IV) {
        byte[] bytebuffer = new byte[36];
        int currIndex = 0;
        int index = 0;

        //load Register A
        for (int i = 0; i < key.length; i++) {

            bytebuffer[currIndex++] = key[i];
        }
```

```
        bytebuffer[currIndex++] = 0;

        //copy the first five bits of the IV
        byte joint = IV[index++];
        bytebuffer[currIndex++] = (byte) ((joint & 0xff) >>> 5);

        //copy the rest of IV to register B
        while (index < IV.length) {
            byte tmp = IV[index - 1];
            byte tmp2 = IV[index];

            tmp = (byte) (tmp << 3);
            tmp2 = (byte) ((tmp & 0xff) >>> 5);
            //bit addition
            int copy = tmp + tmp2;
            //new byte
            bytebuffer[currIndex++] = (byte) copy;
            index++;
        }

        joint = IV[IV.length - 1];
        bytebuffer[currIndex++] = (byte) (joint << 3);

        //Register C
        while (currIndex < bytebuffer.length - 1) {
            bytebuffer[currIndex++] = 0;
        }

        bytebuffer[currIndex++] = 7;

        state = BitSet.valueOf(bytebuffer);
        rotate();

    }//end init method (with key and IV input)

    /* Generates specified number of bytes and update state.*/

    public synchronized void getBytes(byte[] output) {
        boolean t1, t2, t3;
        boolean[] z = new boolean[8];
        int i = 0, index = 0, count = 0, b = output.length * 8;

        boolean s66, s69, s91, s92, s93, s162, s171, s175,
                s176, s177, s243, s264, s286, s287, s288;

        while (count < b) {
            s66 = state.get(65);
            s69 = state.get(68);
            s91 = state.get(90);
            s92 = state.get(91);
            s93 = state.get(92);
```

```
            s162 = state.get(161);
            s171 = state.get(170);
            s175 = state.get(174);
            s176 = state.get(175);
            s177 = state.get(176);
            s243 = state.get(242);
            s264 = state.get(263);
            s286 = state.get(285);
            s287 = state.get(286);
            s288 = state.get(287);
            //generate bit
            t1 = s66 ^ s93;
            t2 = s162 ^ s177;
            t3 = s243 ^ s288;
            z[i] = t1 ^ t2 ^ t3;
            //update
            t1 = t1 ^ s91 & s92 ^ s171;
            t2 = t2 ^ s175 & s176 ^ s264;
            t3 = t3 ^ s286 & s287 ^ s69;
            //shift register A
            for (int j = 91; j >= 0; j--) {
                state.set(j + 1, state.get(j));
            }
            state.set(0, t3); //s1
            //shift register B
            for (int j = 175; j >= 93; j--) {
                state.set(j + 1, state.get(j));
            }
            state.set(93, t1);//s94
            //shift register C
            for (int j = 286; j >= 177; j--) {
                state.set(j + 1, state.get(j));
            }
            state.set(177, t2);//s178

            i++;

            //copy generated byte to state
            if (i == 8) {
                for (int j = 0; j < 8; j++) {
                    if (z[j]) {
                        output[index] |= (byte) (1 << (7 - j));
                        z[j] = false;//reset
                    }
                }
                index++;
                i = 0; //reset z index
            }
            count++;
        }
```

```
}//end getBytes method

/* Rotates the state bits over four cycles. */

private void rotate() {
    boolean t1, t2, t3;
    int i = 0;
    int cycles = (4 * state.length());

    boolean s66, s69, s91, s92, s93, s162, s171, s175,
            s176, s177, s243, s264, s286, s287, s288;

    while (i <= cycles) {
        s66 = state.get(65);
        s69 = state.get(68);
        s91 = state.get(90);
        s92 = state.get(91);
        s93 = state.get(92);
        s162 = state.get(161);
        s171 = state.get(170);
        s175 = state.get(174);
        s176 = state.get(175);
        s177 = state.get(176);
        s243 = state.get(242);
        s264 = state.get(263);
        s286 = state.get(285);
        s287 = state.get(286);
        s288 = state.get(287);

        t1 = s66 ^ s91 & s92 ^ s93 ^ s171;
        t2 = s162 ^ s175 & s176 ^ s177 ^ s264;
        t3 = s243 ^ s286 & s287 ^ s288 ^ s69;

        //shift register A
        for (int j = 91; j >= 0; j--) {
            state.set(j + 1, state.get(j));
        }
        state.set(0, t3); //s1

        //shift register B
        for (int j = 175; j >= 93; j--) {
            state.set(j + 1, state.get(j));
        }
        state.set(93, t1);//s94

        //shift register C
        for (int j = 286; j >= 177; j--) {
            state.set(j + 1, state.get(j));
        }
```

```
            state.set(177, t2);//s178
            i++;
        }
    }//end rotate method

}//end Trivium Class
```

Appendix C

Statistical Test Descriptions

NIST Statistical Test Suite

Frequency (Monobit) Test

The Frequency (Monobit) Test verifies that the existence of zeroes and ones—bit-wise— are equally proportional. In relation to the other tests in the suite, this is by far the most important; if an ensemble cannot pass this test, most likely it will not pass the other fifteen. Remember, one of the assumptions of truly random sequence is uniform distribution of zeroes and ones and another is scalability. Thus the number of zeroes and ones must be balanced bit-wise in order to be balanced word-wise. The test takes a sequence and converts all the zero bits to a value of negative one (-1) and all the one bits to positive one (+1). The values are all added together. This value will be used to compute the test statistic. After the test statistic is determined, it is used to calculate the P-value. If the P-value is greater than or equal to 0.01 the sequence is said to be random.

Frequency Test within a Block

This assessment focuses on the proportions of the 1-bit in the subsequences of a random quantity. The random quantity is first divided into subsequences (or blocks) all of equal length. Leftover bits are discarded. The fraction of ones in each subsequence is calculated. These values are used to determine the chi-squared statistic. Lastly, the P-value is calculated using the chi-squared statistic. Like the Frequency (Monobit) Test, the P-value must be greater than or equal to 0.01 to be declared random.

Runs Test

A run is defined as a chain of identical bits. In this test, a run begins at the first bit whose preceding bit is of opposite value and terminates when the immediately successive bit is of opposite value; its length is the number of bits in the said run. The Runs Test determines if a sequence switches too quickly or too slowly between zeroes and ones. The Frequency test must be performed before conducting the Runs Test. The next step is to calculate the fraction of ones in the sequence. Now, if the sequence passed the Frequency test and the fraction of ones is within a certain boundary described by the literature, there is no need to proceed with this assessment. If the sequence failed to pass this bound, the test statistic is calculated followed by the P-value. The P-value here must also be less than 0.01 to be considered non-random.

Test for the Longest Runs of Ones in a Block

The longest runs of ones in subsequences of identical lengths are identified and recorded. A random quantity is partitioned into sequences of n-length. (Remainder bits are discarded.) Then a table is drawn with the subsequences' length and their longest runs length. Next, determine the chi-square statistic and the P-value. A large chi-square value indicates that the sequence contains clusters of ones [Ruk+10]. A sequence is random if the P-value is greater than or equal to 0.01.

Binary Matrix Rank Test

This assessment is a part of another common PRNG testing suite, Diehard Battery of tests. The Binary Matrix rank test verifies there is linearity within subsequences. The procedure constructs

matrices of consecutive bits in the sequence, and then looks for linearity between the rows and the columns of the matrices. The deviation of the matrices ranks from what is expected from a truly random matrix rank is the main result of determination. The test calculates a chi-square statistic and a P-value to determine if the examined sequence is random. Large value of chi-square denotes that there is a significant difference between the rank distributions of the generated sequence with that of a truly random sequence.

Discrete Fourier Transform (Spectral) Test
This test is based on the discrete Fourier transform, a function that maps an *n*-divided sequence of *m*-length to *n* number *of m*-length sequences in discrete time Fourier transform. They are used to reveal information about periodicities of data as well as the degree he purpose of this test is to identify recurring patterns within close proximity of one another. The test process can be found in the literature.

Non-overlapping Template Matching Test
Blocks of length *n* are identified and checked if their occurrences agree with what is expected in a truly random sequence. This helps to identify if a generator outputs a significant number of irregular patterns. The testing begins by creating an *n*-bit window and aligning it to the first bit of the tested sequence. The window is looking for a specific pattern. If the pattern is not detected, the window shifts one bit to the left. If a pattern is detected, the window shifts *n*-bits to the left. [ZG12]

Overlapping Template Matching Test
This assessment is akin to the Non-overlapping Template Matching test but the difference is the window always shift one bit to the left regardless if a pattern is found or not.

Note: The following three tests involve the notion of compression. Compressibility is the capability of an expression to be simplified to a coherent description or algorithm. Hence, it is desirable for a random sequence to be uncompressible.

Maurer's "Universal Statistical" Test
This test focuses on the number of *n*-bit blocks linking two identical *n*-bit permutations in a sequence. First, a sequence is divided into n-bit blocks and unused bits are discarded. Then, the sequence is separated into two parts: an initialization section (the first set of blocks in the sequence) and a test section (the remaining blocks). The test section contains more blocks than the initialization section. For each section, a table is drawn with columns of all possible n-bit patterns and rows of block indices. The last occurrence of every possible pattern is recorded at each block. Once the tables are completed the cumulative sum is taken; it includes the natural logs of the number of blocks linking two identical patterns. The main function of the test is to determine "if a sequence can be significantly compressed without loss of information" [Ruk+10].

Lempel-Ziv Compression Test [13]
The Lempel-Ziv Compression Test determines to what degree a sequence can be simplified, in other words, compressed. The process begins by making a dictionary of all the unique bit

[13]This test is included in the NIST Statistical Test Suite document but is not a part of the actual program.

patterns in the tested sequence. The total number of distinct patterns, theoretical mean and the variance are used to determine the P-value. If the P-value is less than 0.01, then the sequence is considerably compressible and therefore classified as non-random.

Linear Complexity Test
This evaluation perfectly divides a sequence into *n*-bit blocks and performs the Berlekamp-Massey Algorithm on each block. The algorithm is used to find the shortest LFSR within a block. A very short LFSRs indicate that the sequence is not random and the bits are highly correlated.

The following two tests base their analysis on intersecting patterns of length *n* in a tested sequence.

Serial Test
This test counts and compares the occurrences of 2 *m*-bit overlapping patterns to the expected value of a uniformly distributed sequence. a sequence passes this test if the occurrences is close to the expected value.

Approximate Entropy Test
Approximate entropy "measures the logarithmic frequency with which blocks of length *m* that are close together remain close together for blocks augmented by one position" [PB96]. In other words, in a sequence of length *n*, the frequency of overlapping patterns of *m*-bit length in comparison to overlapping patterns of (*m*+1)-bit length are quantified. Entropy— here—is the repetition of bit patterns. The difference between the entropies of *m*-bits and (*m*+1)-bits must be large to deem the sequence random. Sequences that are random have an approximate entropy value close to ln 2. [Ruk00]

Cumulative Sums Test
The Cumulative Sums Test evaluates if the occurrences of ones and zeroes are mixed evenly throughout the tested sequence. The desired result is a random walk, the cumulative sum of subsequences, is near zero; this suggests that the sequence is random.

Random Excursions Test
This analysis checks if the visits to a particular cumulative sums reflect truly random behavior. There are eight types of cumulative sums ranging from -4 to 4, excluding zero. For each type, a separate test must be given. So in actuality, the Random Excursion Test is a set of eight tests.

Random Excursions Variant Test
Like the Random Excursions test, this assessment also checks the visits to particular cumulative sums. Eighteen tests are performed to assess the cumulative sums from negative nine to positive nine.

Diehard Battery of Test

Birthday Spacings Test
This assessment uses principles from the birthday problem, an argument in probability theory that claims in a room filled with *n* number of individuals, two will have the same birthdays. In relation to the tested sequence, the calendar days are 2^{24} and the number of birthdays is 2^9. The

number of repeated birthdays in a calendar is tabulated. The test produces p-values from the chi-square test and KSTEST.

The Overlapping 5-Permutation Test

This test counts the permutations of a set of five consecutive integers. The integers are all 32 bits in length and are a part of a larger sequence of one million integers. A tally is taken to note the occurrence of each permutation.

Binary Rank Tests

Depending on the test, a number of bits are selected to form a matrix. The rows are then studied to see if they are linearly independent. The maximum number of linearly independent rows is used to determine the rank of the matrix. The ranks of the matrices used in a given test are tallied. A chi-square test is performed on rank counts greater than a certain threshold.

Binary Rank Test: 31x31 matrices

This test requires forty thousand matrices of the said size. Each matrix consists of 31 random integers; each integer of 32-bit length; and the leftmost 31 bits of an integer form a column (or row) in the matrix. A count is taken for the following ranks: ranks less than or equal to 28, ranks of 29, ranks of 30 and ranks of 31. Lastly, the chi-square test is executed on four counts.

Binary Rank Test: 32x32 matrices

Like the aforementioned Binary Rank Test, forty thousand matrices are used in this assessment. Each matrix consists of thirty two 32-bit integers. A chi-square test is performed on counts of ranks less than or equal to 29, ranks of 30, ranks of 31 and ranks of 32.

Binary Rank Test: 6x8 matrices

First, six 32-bit integers are random selected. Then a sequence of 8-bits is selected from each integer. The six bytes are then used to form a matrix for analysis. A hundred thousand matrices are ranked and tallied. A chi-square test is performed on ranks less than or equal to 4, ranks of 5 and ranks of 6.

Bitstream Test

This test counts the number of 20-letter word missing in a stream of 2^{21} overlapping words. A letter is represented by a single bit, 0 or 1.The test is carried out twenty times to determine a p-value.

OPSO, OQSO and DNA Tests

These tests count the number of *n*-letter words missing from the sequence under examination. OPSO is concerned with 2-letter words; OQSO considers 4-letter words and DNA 10-letter words.

Count the 1s Test for Byte-Streams and Specific Bytes

A byte is mapped to a letter on the set {A, B, C, D, E}. Each letter represents a byte with a specific number of ones. A is a byte with zero, one or two 1s; B is three 1s, C is four 1s, D is five 1s and E is six, seven or eight 1s. After the transformation, a combination of 5-letter words is formed with replacement. Next, each word's frequency is totaled. For the byte stream analysis,

the test sequence is parsed into 8-bit subsequences. For the specific bytes analysis, the test sequence is parsed into 32-bit integers. A subsequence of 8 bits is then taken from each integer to carry out the test.

This is a Parking Lot Test

This assessment takes a square of 100 x 100 and tries to randomly park as many unit circles in 12,000 attempts. The number of cars parked without a collision is totaled. A crash is the parking of a car in a spot that is already occupied.

Minimum Distance Test

This assessment takes a square of 10^4 x 10^4 and haphazardly place eight thousand points in the square. Then, the minimum distance between $(n^2 - n)/2$ pairs of points is determined. Here n equals 8000. This process is executed a total of one hundred times to collect a minimum distance value. Finally, KSTEST is performed on the p-values collected.

3DSpheres Test

This assessment takes a cube with length of on each edge. Four thousand points are randomly placed in the cube. Each point is the center of a sphere with the radius being distance to the closest point. The smallest sphere's radius is exponentially distributed. This process is repeated to total twenty times. A KSTEST is performed on the p-values collected from all runs.

Squeeze Test

Random floats on [0, 1) are multiplied by 2^{31}. The test counts how many random floats are used to achieve a value of 1. This procedure is carried out 100,000 times. The counts are tallied and a chi-square test is carried out.

Overlapping Sums Test

First, a long sequence of uniform variables is formed by floated integers. The overlapping sum consists of 100 consecutive uniform variables.

Runs Test

This is a standard test included in the suite. It counts the frequencies of runs, a consecutive sequence of identical bits, forward and backward in an input file.

Craps Test

A game of craps is played 200,000 times. The wins and throws per game is tallied. A chi-square test is performed on the occurrence of the number of throws value.

Dieharder: A Random Number Test Suite

Bit Distribution Test

This test tabulates the frequencies of independent n-bit long sequences. The distribution acquired from the frequencies is compared to the theoretical binomial histogram f a uniform distribution. Standard testing procedures requires the sequence length to be greater than zero but lest than twelve. Sequences longer than twelve requires the sample size to be increased. The p-value is obtained by performing chi-square testing.

Generalized Minimum Distance Test

This test is a generalized version of the 2d sphere test and 3d sphere test available in Diehard battery of tests. This test can be administered using the suggested d (dimension) values of 2, 3, 4 and 5. Comparing to the original versions found in diehard, this test allows control over three parameters that are static in its original form. Analysts can set the parameters for the number of points used for a minimum distance sample; the number of test runs; and the number of dimensions.

Permutations Test

This test counts the permutations of some arbitrary number of integers in the tested sequence. The integers are expected to occur (n!) times. A chi-square test is performed on the count.

Lagged Sums Test

This assessment checks for lagged correlation, a connection between randomly generated sequences on a bit level. The values that deviate from uniform are sampled and summed. The mean and standard deviation the sample is computed, the p-value is then calculated and finally a KSTest is performed on the p-values.

The Kolmogorov-Smirnov Test Test

A Kolmogorov-Smirnov Test of all the values collected from the KSTest of the other tests in the suite.

DAB Byte Distribution Test

In this assessment, three non-overlapping bytes are sampled from three words. These words are consecutive outputs produced from a random number generator. In cases where the words are too small, then the sampled bytes will be overlapping. The p-value is obtained by using a chi-square fitting test.

DCT (Frequency Analysis) Test

This test uses discrete cosine transform (DCT) on samples from a generator. (DCT is a linear function written in terms of a sum of cosines.) These values are collected and used to check for uniformity or independence between samples.

DAB Fill Tree Test

The purpose of this test is to check if a generator shows bias between low, middle or high bytes. The procedure starts by constructing a binary tree of fixed length and inserting subsequences from the output until the tree reaches capacity. The subsequences are rotated a number of times to determine if there is bias.

DAB Fill Tree 2 Test

This test is like the DAB Fill tree test but operates on a bit level. For every bit of the tested sequence, the test takes a step right or left for zero-bits and one-bits, respectively.

I have submitted this thesis in partial fulfillment of the requirements for the degree of Master of Science.

02 May 2017
Date

LaToya Niesha Jackson
Latoya Niesha Jackson

We approve the thesis of Latoya Niesha Jackson as presented here.

May 12, 2017
Date

Yeşem Kurt-Peker
Assistant Professor of Computer Science, Thesis Adviser

May 12, 2017
Date

Radhouane Chouchane
Associate Professor of Computer Science

5/12/2017
Date

Rodrigo Obando
Associate Professor of Computer Science

5/12/2017
Date

Lydia Ray
Lydia Ray
Associate Professor of Computer Science